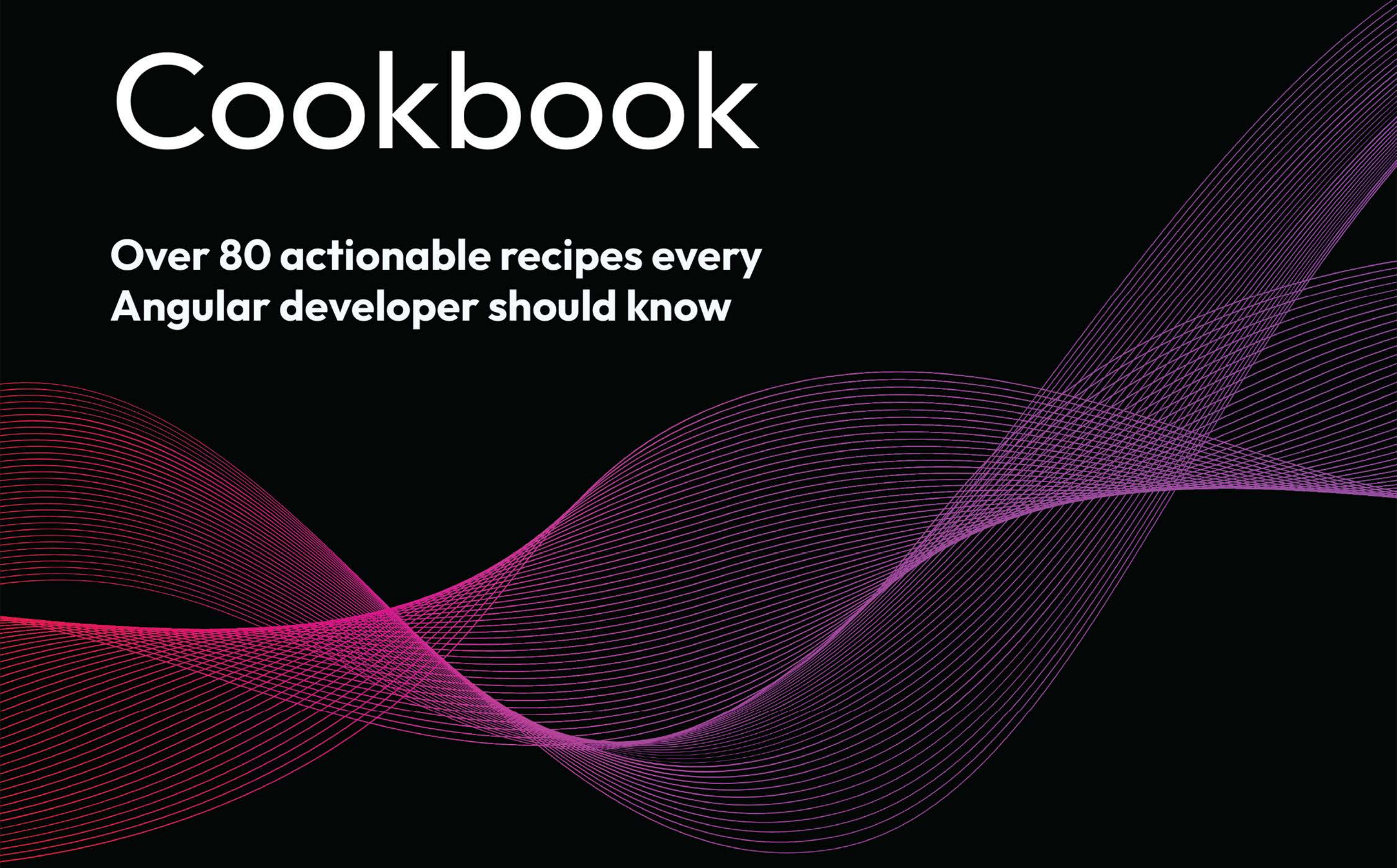
EXPERT INSIGHT

Angular

Angular Cookbook

Over 80 actionable recipes every
Angular developer should know



Second Edition

Muhammad Ahsan Ayaz

packt

Angular Cookbook

Second Edition

Over 80 actionable recipes every Angular developer should know

Muhammad Ahsan Ayaz

packt
BIRMINGHAM—MUMBAI

"Angular" and the Angular Logo are trademarks of the Google LLC

Angular Cookbook

Second Edition

Copyright © 2023 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Senior Publishing Product Manager: Suman Sen
Acquisition Editor – Peer Reviews: Gaurav Gavas
Project Editor: Amisha Vathare
Content Development Editor: Shazeen Iqbal
Copy Editor: Safis Editing
Technical Editor: Anjitha Murali
Proofreader: Safis Editing
Indexer: Subalakshmi Govindhan
Presentation Designer: Ganesh Bhadwalkar
Developer Relations Marketing Executive: Priyadarshini Sharma

First published: August 2021
Second edition: December 2023

Production reference: 1211223

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80323-344-4

`www.packt.com`

Contributors

About the author

Muhammad Ahsan Ayaz is a father, a son, a brother to two awesome siblings, a husband to a caring and supportive wife, a Muslim, a **Google Developers Expert** (**GDE**) in Angular, a software architect, and an educator. Over the past decade, he has trained thousands of developers through free video courses, one-to-one mentoring, and workshops. He has created lots of open-source and free software, including `ngx-device-detector`, which had amassed over 10 million installs by the end of 2023, with over 4500 projects utilizing it solely on GitHub. Ahsan is passionate about assisting the startup ecosystem and product owners in bringing their ideas to life using JavaScript, Angular, and web technologies. Furthermore, Ahsan also runs a community called "Code with Ahsan," boasting a membership of over 25,000 developers worldwide. You can access Ahsan's technical articles and free courses at `https://codewithahsan.dev`, as well as his video tutorials at `https://youtube.com/codewithahsan`.

I want to express my gratitude to the almighty Allah for enabling me to work on this book, so that a significant number of people can benefit from it, and for the fact that I learned a lot while I was working on this book. I also want to extend my heartfelt thanks to the people who have played a pivotal role in my life journey and whatever little I have achieved. Especially my parents (Zahida Khatoon and Muhammad Ayaz), who have always prioritized my needs before their own and raised me to become the person I am today. I also want to thank my wife, Najla, who has supported me throughout the years to continue my passion for educating others and providing guidance during moments when I needed clarity and to make thoughtful decisions. Additionally, I wish to thank Amisha Vathare, Suman Sen, and Shazeen Iqbal, and everyone from Packt who have helped me in ensuring the exceptional quality of the content in this book. Special thanks to Marcell Kiss (`marcell.tech`*) for conducting a comprehensive technical review of the book.*

About the reviewer

Marcell Kiss, MSc, is an independent Angular consultant who advises enterprise clients on maintainable frontend architectures. He also serves as a part-time trainer at Angular Architects and is the organizer of the Angular Hungary Meetup.

I deeply appreciate the endless patience of my family while I was spending long hours reviewing this book.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

Table of Contents

Preface xxi

Chapter 1: Winning Component Communication 1

Technical requirements 1
Component communication using component @Input and @Output properties 2
  Getting ready • 2
  How to do it... • 3
  How it works... • 7
  See also • 7
Component communication using services 7
  Getting ready • 8
  How to do it... • 8
  How it works... • 10
  See also • 11
Using setters for intercepting input property changes 11
  Getting ready • 11
  How to do it... • 12
  How it works... • 15
  See also • 16
Using ngOnChanges to intercept input property changes 16
  Getting ready • 16
  How to do it... • 17
  How it works... • 20
  See also • 20
Accessing a child component in a parent template via template variables 21
  Getting ready • 21
  How to do it... • 22

How it works... • 23
See also • 24
Accessing a child component in a parent component class using ViewChild **24**
Getting ready • 24
How to do it... • 25
How it works... • 26
See also • 26
Standalone components and passing data through route params **26**
Getting ready • 26
How to do it... • 27
How it works... • 30
See also • 30
Component communication using signals **31**
Getting ready • 31
How to do it • 32
How it works • 40
See also • 41

Chapter 2: Working with Angular Directives and Built-In Control Flow 43

Technical requirements **43**
Using attribute directives to handle the appearance of elements **44**
Getting ready • 44
How to do it... • 45
How it works... • 49
See also • 50
Creating a directive to calculate the read time for articles **50**
Getting ready • 50
How to do it... • 51
How it works... • 56
See also • 56
Creating a directive that allows you to vertically scroll to an element **56**
Getting ready • 56
How to do it... • 57
How it works... • 60
See also • 60
Writing your first custom structural directive **60**
Getting ready • 60

  How to do it... • 61
  How it works... • 66
  See also • 67
How to apply multiple structural directives to the same element **67**
  Getting ready • 67
  How to do it... • 68
  How it works... • 70
  See also • 70
Applying multiple directives to the same element using the Directive Composition API **70**
  Getting ready • 70
  How to do it... • 71
  How it works... • 76
  See also • 76

Chapter 3: The Magic of Dependency Injection in Angular 79

Technical requirements **79**
Using Angular DI tokens **79**
  Getting ready • 80
  How to do it... • 80
  How it works... • 81
  See also • 81
Optional dependencies **81**
  Getting ready • 82
  How to do it... • 82
  How it works • 86
  See also • 86
Creating a singleton service using providedIn **86**
  Getting ready • 86
  How to do it • 87
  How it works • 91
  See also • 91
Creating a singleton service using forRoot() **91**
  Getting ready • 92
  How to do it • 92
  How it works • 94
  See also • 95

Providing alternate classes against the same DI Token 95
    Getting ready • 95
    How to do it • 96
    How it works • 97
    See also • 97
Dynamic configurations using value providers 97
    Getting ready • 97
    How to do it • 98
    How it works • 100
    See also • 101

Chapter 4: Understanding Angular Animations 103

Technical requirements 103
Creating your first two-state Angular animation 103
    Getting ready • 104
    How to do it... • 104
    How it works... • 106
    See also • 107
Working with multi-state animations 107
    Getting ready • 108
    How to do it... • 108
    How it works... • 111
    See also • 111
Creating complex Angular animations using keyframes 111
    Getting ready • 112
    How to do it... • 112
    How it works... • 116
    See also • 116
Animating lists in Angular using stagger animations 116
    Getting ready • 116
    How to do it... • 117
    How it works... • 120
    See also • 120
Sequential vs parallel animations in Angular 121
    Getting ready • 121
    How to do it... • 121

How it works... • 125

See also • 125

Route animations in Angular 125

Getting ready • 125

How to do it... • 126

How it works... • 131

See also • 132

Disabling Angular animations conditionally 132

Getting ready • 132

How to do it... • 133

How it works... • 134

See also • 135

Chapter 5: Angular and RxJS — Awesomeness Combined 137

Technical requirements 137

Sequential and parallel http calls in Angular with RxJS 137

Getting ready • 138

How to do it... • 138

How it works... • 141

See also • 142

Listening to multiple observable streams 142

Getting ready • 142

How to do it... • 143

How it works... • 146

See also • 146

Unsubscribing streams to avoid memory leaks 146

Getting ready • 146

How to do it... • 147

How it works... • 149

There's more... • 149

See also • 151

Using Angular's async pipe to unsubscribe streams automatically 152

Getting ready • 152

How to do it... • 152

How it works... • 155

See also • 155

Using the map operator to transform data **155**

Getting ready • 155

How to do it... • 156

How it works... • 159

See also • 159

Using the switchMap and debounceTime operators with autocompletes for better performance ... 159

Getting ready • 159

How to do it... • 160

How it works... • 163

See also • 163

Creating a custom RxJS operator **163**

Getting ready • 163

How to do it... • 164

How it works... • 166

See also • 166

Retrying failed HTTP calls with RxJS **166**

Getting ready • 166

How to do it... • 167

How it works... • 170

See also • 170

Chapter 6: Reactive State Management with NgRx 171

Technical requirements **171**

Creating your first NgRx store with actions and reducers **171**

Getting ready • 172

How to do it... • 172

See also • 176

Using NgRx Store Devtools to debug state changes **176**

Getting ready • 176

How to do it... • 177

How it works... • 178

There's more... • 179

See also • 179

Using NgRx selectors to select and render state in components **180**

Getting ready • 180

How to do it... • 180

How it works... • 183
See also • 183
Using NgRx effects to fetch data from API calls 183
Getting ready • 183
How to do it... • 184
How it works... • 189
See also • 189
Using NgRx Component Store to manage the state of a component 190
Getting ready • 190
How to do it... • 190
How it works... • 195
See also • 195

Chapter 7: Understanding Angular Navigation and Routing 197

Technical requirements 197
Creating routes in an Angular (standalone) app 197
Getting ready • 198
How to do it... • 198
How it works... • 201
See also • 201
Lazily loaded routes in Angular 201
Getting ready • 202
How to do it... • 202
How it works... • 204
See also • 205
Preloading route strategies 205
Getting ready • 205
How to do it... • 206
How it works... • 211
See also • 212
Authorized access to routes using route guards 212
Getting ready • 212
How to do it... • 212
How it works... • 214
See also • 214
Working with route parameters 215
Getting ready • 215

How to do it... • 215
How it works... • 218
See also • 219
Showing a global loader between route changes ... 219
Getting ready • 219
How to do it... • 220
How it works... • 222
See also • 222

Chapter 8: Mastering Angular Forms 223

Technical requirements ... 223
Creating your first template-driven form with validation ... 223
Getting ready • 224
How to do it... • 224
How it works... • 229
See also • 229
Creating your first reactive form with validation ... 229
Getting ready • 229
How to do it... • 230
How it works... • 234
See also • 235
Testing forms in Angular ... 235
Getting ready • 235
How to do it... • 236
How it works... • 241
See also • 242
Server-side validation using asynchronous validator functions ... 242
Getting ready • 242
How to do it... • 243
How it works... • 245
See also • 246
Implementing complex forms with reactive FormArray ... 246
Getting ready • 246
How to do it... • 247
How it works... • 251
See also • 252

Writing your own custom form control using ControlValueAccessor 252
  Getting ready • 252
  How to do it... • 253
  How it works... • 260
  See also • 262

Chapter 9: Angular and the Angular CDK 263

Technical requirements 263
Using virtual scroll for huge lists 264
  Getting ready • 264
  How to do it... • 264
  How it works... • 266
  See also • 267
Keyboard navigation for lists 267
  Getting ready • 267
  How to do it... • 268
  How it works... • 271
  See also • 272
Pointy little popovers with the Overlay API 273
  Getting ready • 273
  How to do it... • 273
  How it works... • 280
  See also • 283
Input coercion utilities from the Angular CDK 284
  Getting ready • 284
  How to do it... • 284
  How it works... • 287
  See also • 287
Using the CDK Drag and Drop API to move items from one list to another 287
  Getting ready • 287
  How to do it... • 288
  How it works... • 294
  See also • 295
Creating a multi-step game with the CDK Stepper API 295
  Getting ready • 295
  How to do it... • 296

How it works... • 306
See also • 307
Accessible listbox interactions using CDK Listbox directives 307
Getting ready • 307
How to do it... • 308
How it works... • 310
See also • 310
Working with nested menus using the Angular CDK Menu API 310
Getting ready • 311
How to do it... • 311
How it works... • 317
See also • 318

Chapter 10: Writing Unit Tests in Angular with Jest 319

Technical requirements 319
Setting up unit tests in Angular with Jest 320
Getting ready • 320
How to do it... • 321
Bonus: Migrating to Jest in Angular v16 • 324
See also • 325
Providing global mocks for Jest 325
Getting ready • 325
How to do it... • 326
How it works... • 329
See also • 330
Mocking Angular services using stubs 330
Getting ready • 331
How to do it... • 331
How it works... • 333
See also • 334
Using spies on an injected service in a unit test 334
Getting ready • 334
How to do it... • 335
How it works... • 336
See also • 337
Mocking child components and directives using the ng-mocks package 337
Getting ready • 337

How to do it... • 338
How it works... • 343
See also • 343

Writing even easier tests with Angular CDK component harnesses 344
Getting ready • 344
How to do it... • 345
How it works... • 348
See also • 349

Unit-testing responses from HTTP calls 350
Getting ready • 350
How to do it... • 351
How it works... • 355
See also • 355

Unit-testing Angular pipes 355
Getting ready • 356
How to do it... • 356
See also • 358

Chapter 11: E2E Tests in Angular with Cypress 359

Technical requirements 359

Writing your first Cypress test 359
Getting ready • 360
How to do it... • 360
How it works... • 362
See also • 363

Validating if a DOM element is visible on the view 363
Getting ready • 363
How to do it... • 364
How it works... • 368
See also • 368

Testing form inputs and submission 368
Getting ready • 368
How to do it... • 369
How it works... • 373
See also • 374

Waiting for XHRs to finish 374
Getting ready • 374

How to do it... • 375
How it works... • 381
See also • 382
Using Cypress bundled packages **382**
Getting ready • 382
How to do it... • 383
How it works... • 388
See also • 389
Using Cypress fixtures to provide mock data **389**
Getting ready • 390
How to do it... • 390
How it works... • 396
See also • 397

Chapter 12: Performance Optimization in Angular 399

Technical requirements **399**
Using OnPush change detection to prune component subtrees **400**
Getting ready • 400
How to do it... • 401
How it works... • 405
See also • 405
Detaching the change detector from components **406**
Getting ready • 406
How to do it... • 407
How it works... • 415
See also • 415
Running async events outside Angular with runOutsideAngular **416**
Getting ready • 416
How to do it... • 417
How it works... • 424
See also • 425
Using trackBy for lists with *ngFor **425**
Getting ready • 425
How to do it... • 426
How it works... • 428
See also • 429

Moving heavy computation to pure pipes **429**
  Getting ready • 429
  How to do it... • 430
  How it works... • 435
  See also • 436
Using web workers for heavy computation **436**
  Getting ready • 436
  How to do it... • 437
  How it works... • 441
  See also • 443
Using performance budgets for auditing **443**
  Getting ready • 443
  How to do it... • 444
  How it works... • 446
  See also • 447
Analyzing bundles with webpack-bundle-analyzer **447**
  Getting ready • 447
  How to do it... • 447
  See also • 450

Chapter 13: Building PWAs with Angular 453

Technical requirements **453**
Converting an existing Angular app into a PWA with the Angular CLI **453**
  Getting ready • 454
  How to do it... • 455
  How it works... • 459
  See also • 460
Modifying the theme color for your PWA **460**
  Getting ready • 461
  How to do it... • 461
  See also • 463
Using dark mode in your PWA **463**
  Getting ready • 464
  How to do it... • 466
  See also • 471

Providing a custom installable experience in your PWA **472**

Getting ready • 472

How to do it... • 473

How it works... • 480

See also • 480

Precaching requests using an Angular service worker **480**

Getting ready • 480

How to do it... • 481

How it works... • 483

See also • 483

Creating an App Shell for your PWA **484**

Getting ready • 484

How to do it... • 485

How it works... • 489

See also • 490

Other Books You May Enjoy 495

Index 499

Preface

Angular is one of the most popular frameworks in the world for building not only web applications but even mobile and desktop applications as well. Backed by Google and used by Google, this framework is used by millions of applications. Although the framework is well-suited for applications of any scale, enterprises particularly favor Angular due to its opinionated nature and its consistent ecosystem, which includes all the tools necessary for creating modern & production-ready web applications.

While learning the core technologies such as JavaScript, HTML, and CSS is an absolute essential to progress as a web developer, when it comes to a framework, learning the core concepts of the framework itself is important too. When we're dealing with Angular, there are a lot of amazing things we can do with our web applications by learning about, and using, the right tools in the Angular ecosystem. That's where this book comes in.

This book was written for intermediate and advanced Angular developers to polish their Angular development skills with recipes that you can follow easily, play around with, and practice your own variations of. You'll not only learn from the recipes themselves but also from the actual real-life projects associated with the recipes. So, there are a lot of hidden gems in these recipes and projects for you.

Happy coding!

Who this book is for

The book is for intermediate-level Angular web developers looking for actionable solutions to common problems in Angular enterprise development. Mobile developers using Angular technologies will also find this book useful. Working experience with JavaScript and TypeScript is necessary to understand the topics covered in this book more effectively.

What this book covers

Chapter 1, Winning Component Communication, explains different techniques to use to implement communication between components in Angular. `@Input()` and `@Output()` decorators, services, and lifecycle hooks are covered. The chapter also covers Angular Signals, which became stable in Angular v17.

Chapter 2, Working with Angular Directives and Built-In Control Flow introduces Angular directives and some recipes that use attribute directives, structural directives, and the Directive Composition API.

Chapter 3, The Magic of Dependency Injection in Angular, includes recipes that cover optional dependencies, configuring an injection token, using the `providedIn: 'root'` metadata for Angular services, value providers, and aliased class providers.

Chapter 4, *Understanding Angular Animations*, contains recipes for implementing multi-state animations, staggering animations, keyframe animations, and animations for switching between routes in your Angular apps.

Chapter 5, *Angular and RxJS – Awesomeness Combined*, covers recipes on RxJS for executing sequential and parallel HTTP calls. It also has some recipes on the usage of the `combineLatest`, `flatMap`, `debounceTime`, and `switchMap` operators and covers some tips and tricks about using RxJS streams.

Chapter 6, *Reactive State Management with NgRx*, has recipes concerning the famous NgRX library and its core concepts. It covers core concepts such as NgRx actions, reducers, selectors, and effects and looks at using packages such as `@ngrx/store-devtools` and `@component/store`.

Chapter 7, *Understanding Angular Navigation and Routing*, explores recipes on lazily loaded routes, route guards, route preloading strategies, and some interesting techniques to be used with the Angular router.

Chapter 8, *Mastering Angular Forms*, covers recipes for template-driven forms, reactive forms, form validation, testing forms, form arrays, and creating your own form control.

Chapter 9, *Angular and the Angular CDK*, has a lot of cool Angular CDK recipes, including ones on virtual scroll, keyboard navigation, the overlay API, the CDK Menu, CDK drag and drop, the CDK stepper API, and the CDK Listbox API.

Chapter 10, *Writing Unit Tests in Angular with Jest*, covers recipes for unit testing with Jest, exploring global mocks in Jest, mocking services/child components/pipes, using Angular CDK component harnesses, and unit testing Observables.

Chapter 11, *E2E Tests in Angular with Cypress*, has recipes on E2E testing with Cypress in Angular apps. It covers validating forms, waiting for XHR calls, mocking HTTP call responses, using bundled packages with Cypress, and using fixtures in Cypress.

Chapter 12, *Performance Optimization in Angular*, contains some cool techniques to improve an Angular app's performance by using the `OnPush` change detection strategy, lazily loading feature routes, detaching the change detector from a component, using web workers with Angular, using pure pipes, adding performance budgets to an Angular app, and using the `webpack-bundle` analyzer.

Chapter 13, *Building PWAs with Angular*, contains recipes to create a PWA with Angular. It covers specifying a theme color for the PWA, using a device's dark mode, providing a custom PWA install prompt, precaching requests using Angular's service worker, and using App Shell.

To get the most out of this book

The recipes of this book are built with Angular v17 and Angular follows semantic versioning for their releases. Since Angular is constantly being improved, for the sake of stability, the Angular team has provided a predictable release cycle for updates. The release frequency is as follows:

- A major release every 6 months.
- 1 to 3 minor releases for each major release.
- A patch release and pre-release (next or rc) build almost every week.

Source: `https://angular.io/guide/releases#release-frequency`

Software covered in the book	Operating System requirements
Angular v17	Windows, macOS, or Linux
TypeScript 5.x+	
Nx v17	

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Once you've finished reading the book, make sure to tweet to to let me know your feedback about the book. In addition, you can modify the code provided with this book to your taste, upload it to your GitHub repository, and share it. I'll make sure to retweet it:)

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Angular-Cookbook-2E`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: `https://packt.link/gbp/9781803233444`.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Now, we'll move the code from the `the-amazing-list-component.html` file to the `the-amazing-list-item.component.html` file for the item's markup."

A block of code is set as follows:

```
openMenu($event, itemTrigger) {
    if ($event) {
      $event.stopImmediatePropagation();
    }
    this.popoverMenuTrigger = itemTrigger;
    this.menuShown = true;
  }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
.menu-popover {
  ...
  &::before {...}
  &--up {
    transform: translateY(-20px);
    &::before {
      top: unset !important;
      transform: rotate(180deg);
      bottom: -10px;
    }
  }
  &__list {...}
}
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "You will notice that we can't see the entirety of the content of the input—this is somewhat annoying at the best of times because you can't really review it before pressing the **Action** button."

IMPORTANT NOTES

Appear like this.

TIPS

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share your thoughts

Once you've read *Angular Cookbook - Second Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/9781803233444

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Winning Component Communication

In this chapter, you'll master component communication in Angular. You'll learn different techniques to establish communication between components and will learn which technique is suitable in which situation. You'll also learn about the new Signals API, which is stable since Angular v17, in this chapter.

The following are the recipes we're going to cover in this chapter:

- Component communication using component **@Input** and **@Output** properties
- Component communication using services
- Using setters for intercepting input property changes
- Using **ngOnChanges** to intercept input property changes
- Accessing a child component in a parent template via template variables
- Accessing a child component in a parent component class using **ViewChild**
- Standalone components and passing data through route params
- Component communication using signals

Technical requirements

For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter01`.

Component communication using component @Input and @Output properties

You'll start with an app with a parent component and two child components. You'll then use Angular `@Input` and `@Output` decorators to establish communication between them using attributes and **EventEmitter**(s). The flow of the communication is shown in *Figure 1.1*.

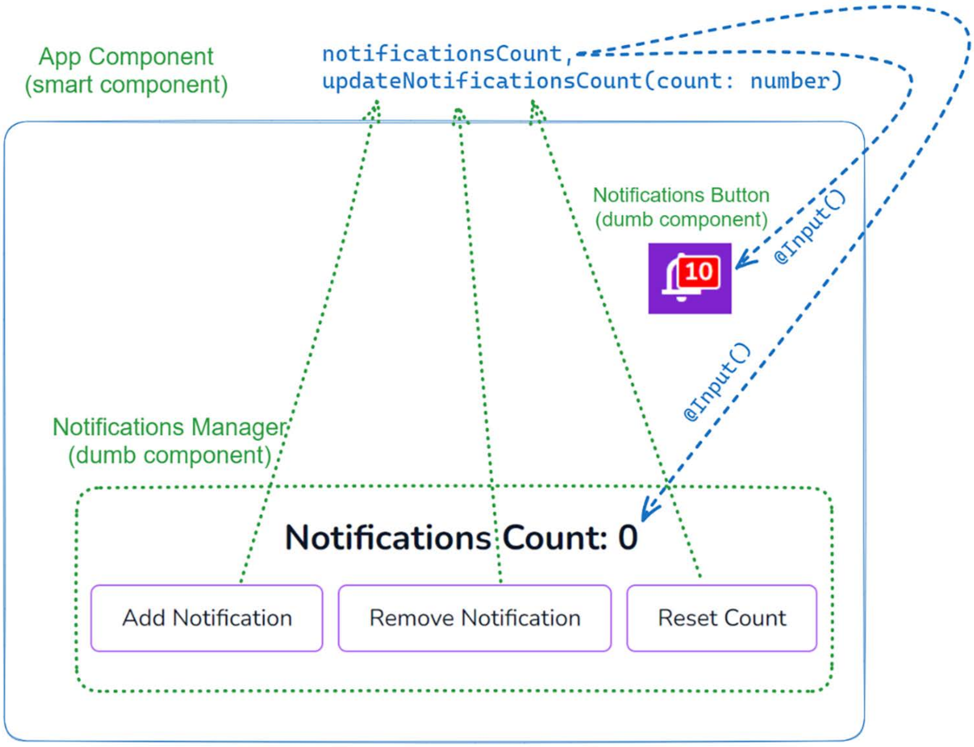

App Component
(smart component)
notificationsCount,
updateNotificationsCount(count: number)
Notifications Button
(dumb component)
10
@Input()
@Input()
Notifications Manager
(dumb component)
Notifications Count: 0
Add Notification
Remove Notification
Reset Count

Figure 1.1: Communication flow using @Input() and @Output() properties

Getting ready

The app that we are going to work with resides in `start/apps/chapter01/cc-inputs-outputs` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve cc-inputs-outputs
```

This should open the app in a new browser tab, and you should see the following:

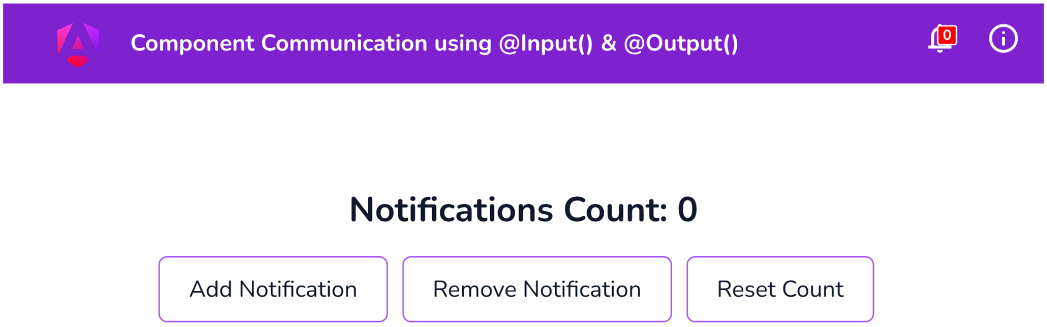


Figure 1.2: The cc-inputs-outputs app running on http://localhost:4200

How to do it...

So far, we have an app with `AppComponent`, `NotificationsButtonComponent`, and `NotificationsManagerComponent`. While `AppComponent` is the parent of the other two components mentioned, there is absolutely no component communication between them to sync the notification count value among both components. Let's establish the appropriate communication between them using the following steps:

1. We'll move the `notificationsCount` variable from `NotificationsManagerComponent` to the `AppComponent`. To do so, create a `notificationsCount` property in `app.component.ts` as follows:

```
export class AppComponent {
  notificationsCount = 0;
}
```

2. Next, convert the `notificationsCount` property in `notifications-manager.component.ts` to `@Input()`, rename it to `count`, and replace its usages as follows:

```
import { Component, OnInit, Input } from '@angular/core';
@Component({
  selector: 'app-notifications-manager',
  templateUrl: './notifications-manager.component.html',
  styleUrls: ['./notifications-manager.component.scss']
})
...
export class NotificationsManagerComponent implements OnInit {
  @Input() count = 0
  constructor() { }
  ngOnInit(): void {
  }
  addNotification() {
    this.count++;
  }
  removeNotification() {
    if (this.count == 0) {
      return;
    }
    this.count--;
  }
  resetCount() {
    this.count = 0;
  }
}
```

3. Update `notifications-manager.component.html` to use `count` instead of `notificationsCount`:

```
<div class="notif-manager">
  <div class="notif-manager__count">
    Notifications Count: {{count}}
  </div>
  ...
</div>
```

4. Next, pass the `notificationsCount` property from `app.component.html` to the `<app-notifications-manager>` element as an input:

```
<div class="content" role="main">
  <app-notifications-manager
    [count]="notificationsCount">
  </app-notifications-manager>
</div>
```

You could now test whether the value is being correctly passed from `app.component.html` to `app-notifications-manager` by assigning the value of `notificationsCount` in `app.component.ts` as `10`. You'll see that in `NotificationsManagerComponent`, the initial value shown will be `10`:

```
export class AppComponent {
  notificationsCount = 10;
}
```

5. Now, create an `@Input()` in `notifications-button.component.ts` named count as well:

```
import { Component, OnInit, Input } from '@angular/core';
...
export class NotificationsButtonComponent implements OnInit {
  @Input() count = 0;
  ...
}
```

6. Pass `notificationsCount` to `<app-notifications-button>` as well from `app.component.html`:

```
<!-- Toolbar -->
<div class="toolbar" role="banner">
  ...
  <span>@Component Inputs and Outputs</span>
  <div class="spacer"></div>
  <div class="notif-bell">
    <app-notifications-button
    [count]="notificationsCount">
    </app-notifications-button>
  </div>
</div>
...
```

7. Use the `count` input in `notifications-button.component.html` with the notification bell icon:

```
<div class="bell">
  <i class="material-icons">notifications</i>
  <div class="bell__count">
    <div class="bell__count__digits">
      {{count}}
    </div>
  </div>
</div>
```

You should now see the value `10` for the notification bell icon count as well.

Right now, if you change the count by adding/removing notifications from `NotificationsManagerComponent`, the count on the notification bell icon won't change.

8. To communicate the change from `NotificationsManagerComponent` to `NotificationsButtonComponent`, we'll use an Angular `@Output()` property now. Use `@Output` and `@EventEmitter` from `'@angular/core'` inside `notifications-manager.component.ts`:

```
import { Component, OnInit, Input, Output, EventEmitter } from '@angular/
core';
...
export class NotificationsManagerComponent implements OnInit {
  @Input() count = 0
  @Output() countChanged = new EventEmitter<number>();
...
  addNotification() {
    this.count++;
    this.countChanged.emit(this.count);
  }
  removeNotification() {
    ...
    this.count--;
    this.countChanged.emit(this.count);
  }
  resetCount() {
    this.count = 0;
    this.countChanged.emit(this.count);
  }
}
```

9. Then, we'll listen in `app.component.html` for the previously emitted event from `NotificationsManagerComponent` and update the `notificationsCount` property accordingly:

```
<div class="content" role="main">
  <app-notifications-manager
    (countChanged)="updateNotificationsCount($event)"
    [count]="notificationsCount">
  </app-notifications- manager>
</div>
```

10. Since we've listened to the `countChanged` event previously and called the `updateNotificationsCount` method, we need to create this method in `app.component.ts` and update the value of the `notificationsCount` property accordingly:

```
export class AppComponent {
  notificationsCount = 10;
  updateNotificationsCount(count: number) {
    this.notificationsCount = count;
  }
}
```

How it works...

In order to communicate between components using `@Input()` and `@Output()` properties, the data flow will always go *from* the **child components** *to* the **parent component via output event emitters** (`@Output()`), **and from the parent component to the child component via input bindings** (`@Input()`). In general, when two sibling components have to communicate, one of them has to use an output emitter to pass the value to the parent, which can then provide the new (updated) value *as an input* to the other child components. So, `NotificationsManagerComponent` emits the `countChanged` event. `AppComponent` (being the parent component) listens for the event and updates the value of `notificationsCount`, which automatically updates the `count` property in `NotificationsButtonComponent` because `notificationsCount` is being passed as the `@Input()` count to `NotificationsButtonComponent`. *Figure 1.3* shows the entire process:

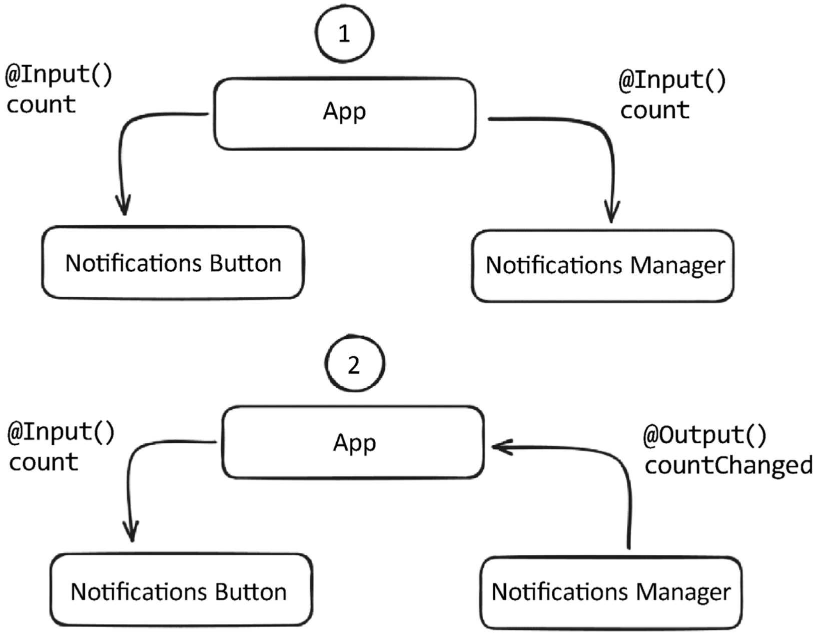


Figure 1.3: How component communication works with inputs and outputs

See also

- How do Angular components communicate? `https://www.thirdrocktechkno.com/blog/how-angular-components-communicate`
- *Component Communication in Angular* by Dhananjay Kumar: `https://www.youtube.com/watch?v=I8Z8g9APaDY`

Component communication using services

In this recipe, you'll start with an app with a parent component and a child component. You'll then use an Angular service to establish communication between them. We're going to use `BehaviorSubject` and `Observable` streams to communicate between components and the service.

Getting ready

The project for this recipe resides in `start/apps/chapter01/cc-services`:

1. Open the project in Visual Studio Code.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve cc-services
   ```

   This should open the app in a new browser tab, and you should see the app as follows:

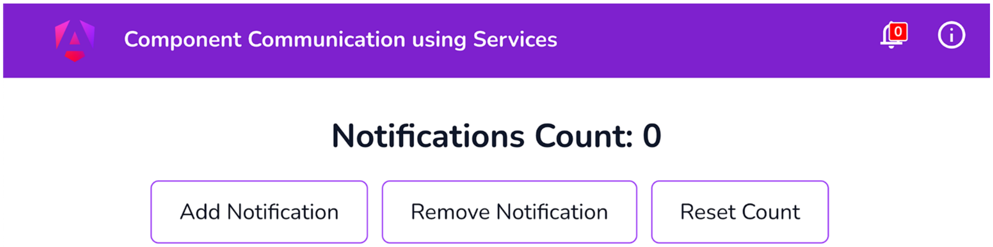


Figure 1.4: The cc-services app running on http://localhost:4200

How to do it...

Similar to the previous recipe, we have an app with `AppComponent`, `NotificationsButtonComponent`, and `NotificationsManagerComponent`. `AppComponent` is the parent of the other two components mentioned previously, and we need to establish the appropriate communication between them using the following steps:

1. From the terminal, make sure you are at the root of the workspace, and create a new service called `NotificationsService` by running the following command:

   ```
   cd start && nx g s services/Notifications --project cc-services
   ```

2. Create a `BehaviorSubject` named `count$` inside `notifications.service.ts` and initialize it with `0` (a `BehaviorSubject` requires an initial value):

   ```
   import { Injectable } from '@angular/core';
   import { BehaviorSubject } from 'rxjs';
   @Injectable({
     providedIn: 'root'
   })
   export class NotificationsService {
     count$ = new BehaviorSubject(0);

     setCount(value: number) {
       this.count$.next(value);
     }
   }
   ```

3. Rename the `notificationsCount` property in `notifications-manager.component.ts` to `notificationsCount$` and assign the services's `count$` property to it as follows:

```
import { Component, inject } from '@angular/core';
import { NotificationsService } from '../services/notifications.service';
...
export class NotificationsManagerComponent implements OnInit {
  notificationService = inject(NotificationsService);
  notificationsCount$ = this.notificationService.count$;
...
}
```

4. Modify the functions with the `NotificationsManagerComponent` to update the `BehaviorSubject` value as follows:

```
...
export class NotificationsManagerComponent implements OnInit {
  ...
  addNotification() {
    const currentValue =
      this.notificationsCount$.getValue();
    this.notificationService.setCount(currentValue + 1);
  }
  removeNotification() {
    const currentValue =
      this.notificationsCount$.getValue();
    if (currentValue === 0) {
      return;
    }
    this.notificationService.setCount(currentValue - 1);
  }
  resetCount() {
    this.notificationService.setCount(0);
  }
}
```

5. Use the `notificationsCount$` Observable in `notifications-manager.component.html` with the `async` pipe to show its value:

```
<div class="notif-manager">
  <div class="notif-manager__count">
    Notifications Count: {{notificationsCount$ | async}}
  </div>
  ...
</div>
```

6. Now, similarly inject `NotificationsService` in `notifications-button.component.ts`, create an Observable named `notificationsCount$` inside `NotificationsButtonComponent`, and assign the service's `count$` Observable to it:

```
import { Component, inject } from '@angular/core';
import { NotificationsService } from '../services/notifications.service';
...
export class NotificationsButtonComponent {
  notificationsCount$ = inject(NotificationsService).count$;
}
```

7. Use the `notificationsCount$` Observable in `notifications-button.component.html` with the `async` pipe:

```
<div class="bell">
  <i class="material-icons">notifications</i>
  <div class="bell__count">
    <div class="bell__count__digits">
      {{notificationsCount$ | async}}
    </div>
  </div>
</div>
```

   If you refresh the app now, you should be able to see the value `0` for both the notifications manager component and the notifications button component.

8. Change the initial value for the `count BehaviorSubject` to `10` and see whether that reflects in both components:

```
...
export class NotificationsService {
  private count: BehaviorSubject<number> = new
    BehaviorSubject<number>(10);
  ...
}
```

How it works...

`BehaviorSubject` is a special type of **Observable** that requires an initial value and can be used by many subscribers. In this recipe, we create a `BehaviorSubject` to store the value of the notifications count.

Once we have created the `BehaviorSubject` named `count$`, we inject `NotificationsService` in our components using the (fairly new) `inject` function and assign the `count$` property of the service to a property of the components. This allows us to work with the `BehaviorSubject` in both the `NotificationsButtonComponent` and the `NotificationsManagerComponent`.

Then we use the `notificationsCount$` property in the templates of both above-mentioned functions to be able to render the count value. Notice that we use the **async** pipe in the templates. This helps Angular to let the template subscribe to the `BehaviorSubject` when the component is rendered and to unsubscribe automatically when the component is destroyed.

To update the value of the `BehaviorSubject`, we use its `next` method by providing the new value to be set. As soon as the value of `count$` is updated, the components rerender the new value, thanks to RxJS and Angular's change detection.

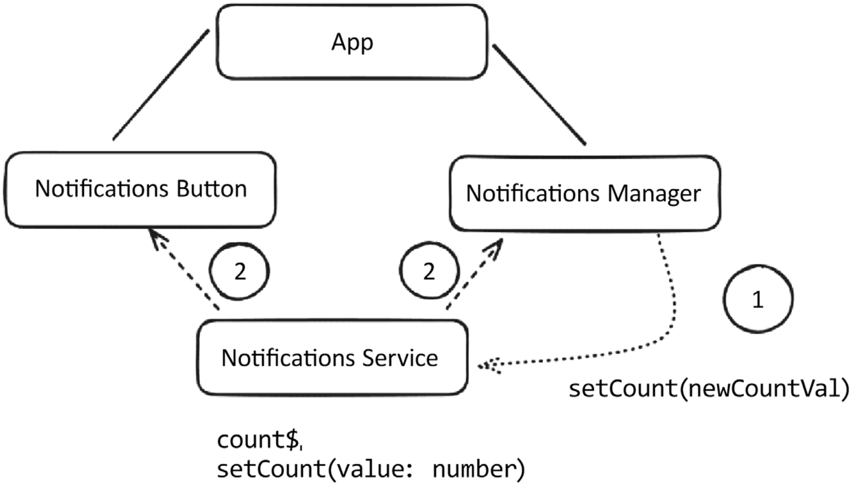


Figure 1.5: How component communication works using an Angular service

See also

- Subjects from RxJS's official documentation: `https://www.learnrxjs.io/learn-rxjs/subjects`
- `BehaviorSubject` versus **Observable** on Stack Overflow: `https://stackoverflow.com/a/40231605`

Using setters for intercepting input property changes

In this recipe, you will learn how to intercept changes in an `@Input` passed from a parent component and how to perform some action on this event. We'll intercept the `vName` input passed from the `VersionControlComponent` parent component to the `VcLogsComponent` child component. We'll use *setters* to generate a log whenever the value of `vName` changes and will show those logs in the child component.

Getting ready

The project for this recipe resides in `start/apps/chapter01/cc-setters`:

1. Open the project in Visual Studio Code.

2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve cc-setters
```

This should open the app in a new browser tab, and you should see the app as follows:

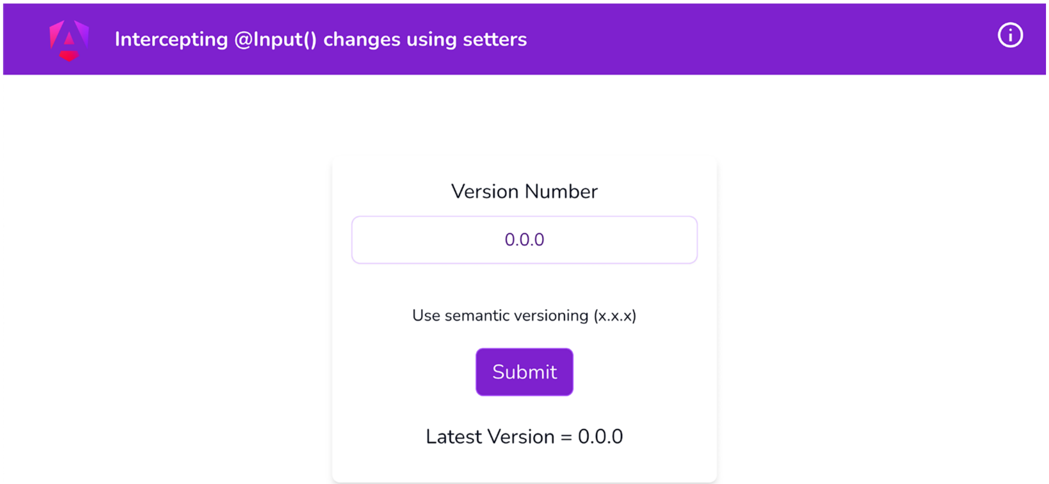

Intercepting @Input() changes using setters
Version Number
0.0.0
Use semantic versioning (x.x.x)
Submit
Latest Version = 0.0.0

Figure 1.6: The cc-setters app running on http://localhost:4200

How to do it...

1. We'll first create a `logs` array in `VcLogsComponent` as follows to store all the logs that we'll display later using our template:

```
export class VcLogsComponent implements OnInit {
  @Input() vName;
  logs: string[] = [];
}
```

2. Let's modify the HTML template so we can show the logs. Modify the `vc-logs.component.html` file as follows:

```
<h5>Latest Version = {{vName}}</h5>
<div class="logs">
  <div class="logs__item" *ngFor="let log of logs">
    {{log}}
  </div>
</div>
```

The following screenshot shows the app with the logs container:

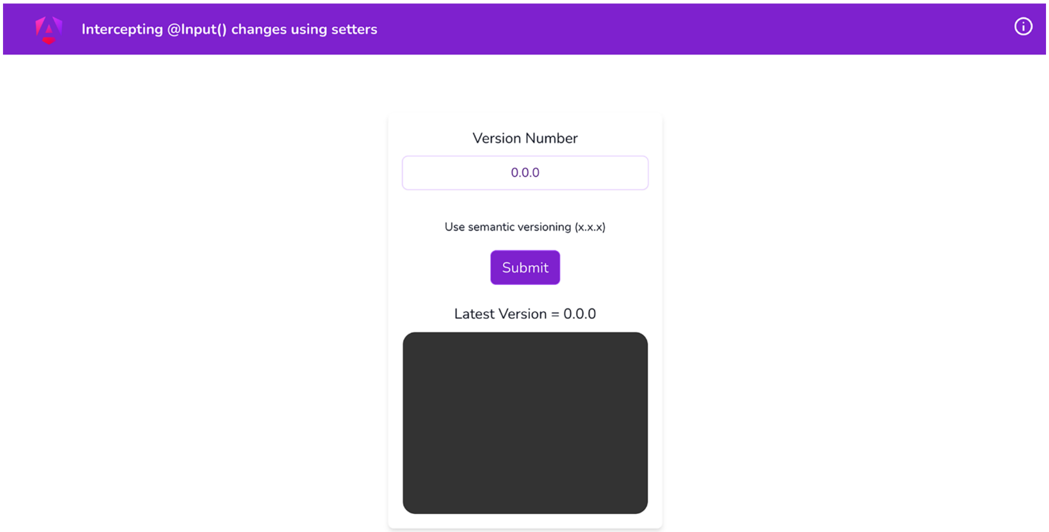

Intercepting @Input() changes using setters
Version Number
0.0.0
Use semantic versioning (x.x.x)
Submit
Latest Version = 0.0.0

Figure 1.7: The cc-setters app with logs container

3. Now, we'll convert the `@Input()` in `vc-logs.component.ts` to use a *getter* and *setter* so we can intercept the input changes. To do that, we'll also create an internal property named `_vName`. The code should look as follows:

```
import { Component, Input } from '@angular/core'
...
export class VcLogsComponent implements OnInit {
  @Input()
  get vName() {
    return this._vName;
  }
  set vName(name: string) {
    this._vName = name;
  }
  logs: string[] = [];
  _vName!: string;
...
}
```

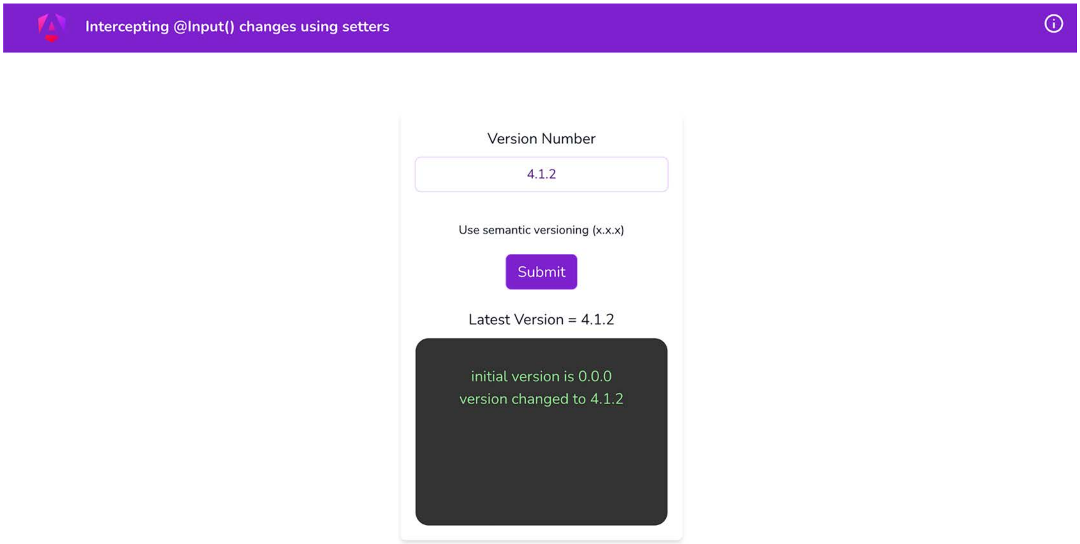
4. Modify the `setter` to create some logs now. We'll push a new log to the `logs` array whenever the value of `vName` changes. For the first time, we'll push a log saying `'initial version is x.x.x'`:

   ```
   export class VcLogsComponent implements OnInit {
     ...
     set vName(name: string) {
       if (!this._vName) {
         this.logs.push(`initial version is ${name.trim()}`)
       }
       this._vName = name;
     }
   ...
   }
   ```

5. Now, every time we change the version name, we need to show a different message saying `'version changed to x.x.x'`. *Figure 1.8* shows the final output. To make the required changes, let's modify the `vName` setter as follows:

   ```
   export class VcLogsComponent implements OnInit {
     ...
     set vName(name: string) {
       if (!name) return;
       if (!this._vName) {
         this.logs.push(`initial version is ${name.trim()}`)
       } else {
         this.logs.push(`version changed to ${name.trim()}`)
       }
       this._vName = name;
     }
   ```

   The following screenshot shows the final output:


Intercepting @Input() changes using setters
Version Number
4.1.2
Use semantic versioning (x.x.x)
Submit

Latest Version = 4.1.2

```
initial version is 0.0.0
version changed to 4.1.2
```

Figure 1.8: Final output using the setter

How it works…

JavaScript has `getters` as functions that return a dynamically computed value. It also has `setters` as functions that execute some logic when the targeted property changes. Angular uses TypeScript, which is a superset of JavaScript and Angular's `@Input()` properties can also use `getters` and `setters` since they're basically properties of the provided class.

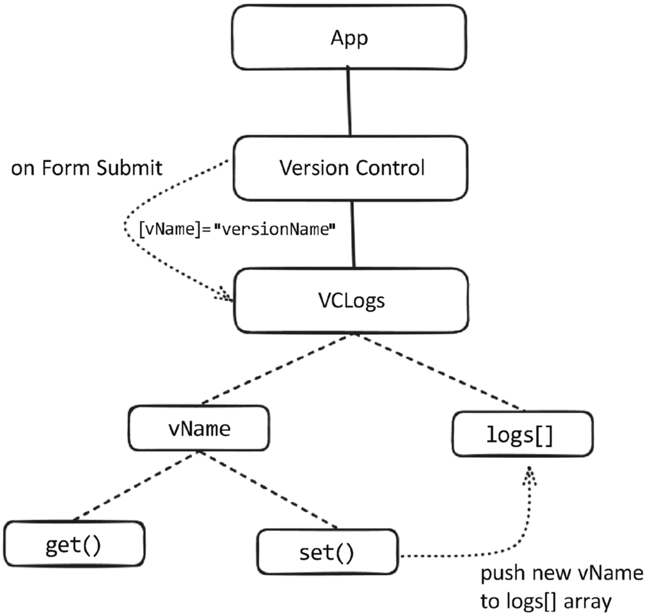

App
on Form Submit
Version Control
[vName]="versionName"
VCLogs
vName
logs[]
get()
set()
push new vName
to logs[] array

Figure 1.9: Diagram explaining the flow of data in the cc-setters app

For this recipe, we use a getter and a setter for our input named `vName` so whenever the input changes, we use the setter function to push the new version to the logs list. Then we use the `logs` array in the template to render the list of logs on the view.

It is always a good idea to use a private variable/property along with the property using getters and setters. This is so that we can modify the private property in our component and the template only accesses the public property using the getter.

See also

- Angular's official docs for setters: `https://angular.io/guide/component-interaction#intercept-input-property-changes-with-a-setter`
- Mozilla's official docs for a getter function: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get`
- Mozilla's official docs for a setter function: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set`

Using ngOnChanges to intercept input property changes

In this recipe, you'll learn how to use `ngOnChanges` to intercept changes using the `SimpleChanges` API. We'll listen to a `vName` input passed from the `VersionControlComponent` parent component to the `VcLogsComponent` child component.

Getting ready

The app that we are going to work with resides in `start/apps/chapter01/cc-ng-on-changes` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve cc-ng-on-changes
```

This should open the app in a new browser tab, and you should see the following:

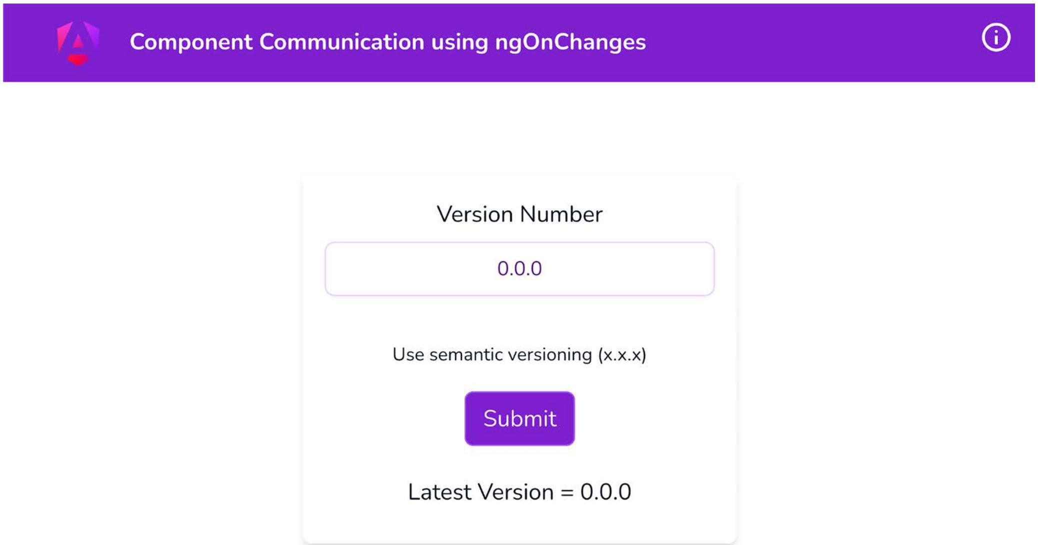

Component Communication using ngOnChanges
Version Number
0.0.0
Use semantic versioning (x.x.x)
Submit
Latest Version = 0.0.0

Figure 1.10: The cc-ng-on-changes app running on http://localhost:4200

How to do it...

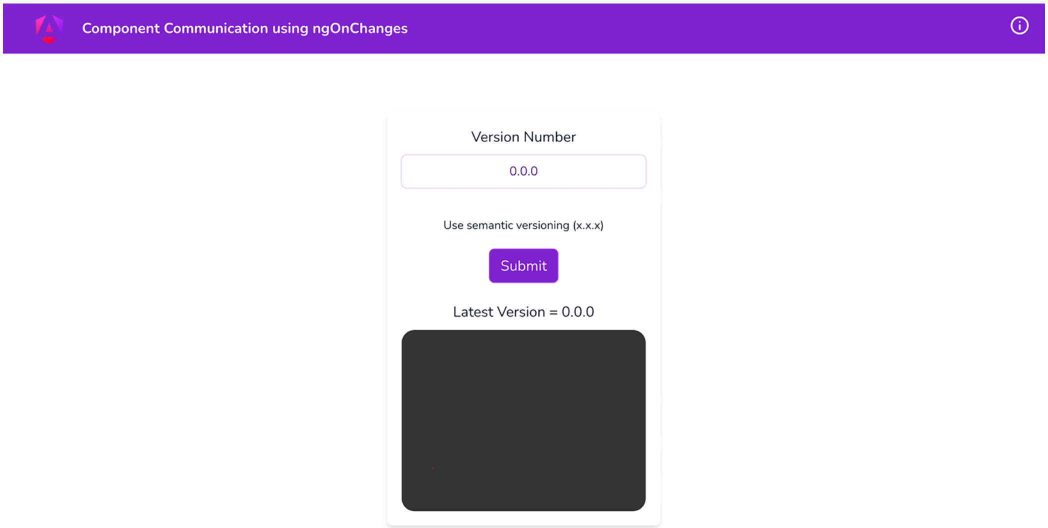
1. We'll first create a logs array in `VcLogsComponent` as follows to store all the logs that we'll display later using our template:

```
export class VcLogsComponent implements OnInit {
  @Input() vName;
  logs: string[] = [];
...
}
```

2. Let's create the HTML for where we'll show the logs. Let's add the `logs` container and log items using the following code to `vc-logs.component.html`:

```
<h5>Latest Version = {{vName}}</h5>
<div class="logs">
  <div class="logs__item" *ngFor="let log of logs">
    {{log}}
  </div>
</div>
```

The following screenshot shows the app with the `logs` container:



Component Communication using ngOnChanges

Version Number

0.0.0

Use semantic versioning (x.x.x)

Submit

Latest Version = 0.0.0

Figure 1.11: The cc-ng-on-changes app with logs container

3. Now, let's implement `ngOnChanges` using simple changes in `VcLogsComponent` as follows in the `vc-logs.component.ts` file:

```
import { Component, Input, OnChanges, SimpleChanges } from '@angular/
core';
...
export class VcLogsComponent implements OnChanges {
  @Input() vName;
  logs: string[] = [];
  ngOnChanges(changes: SimpleChanges) {
  }
}
```

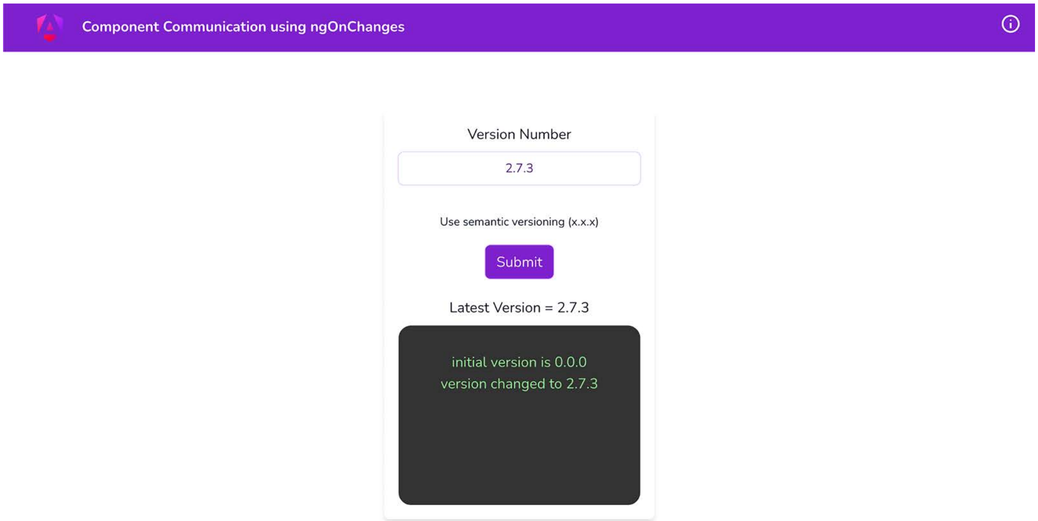
4. We now can add a log for the initial value of the `vName` input saying `'initial version is x.x.x'`. We do this by checking whether it is the initial value using the `isFirstChange` method as follows:

```
...
export class VcLogsComponent implements OnChanges {
  ...
  ngOnChanges(changes: SimpleChanges) {
    const { currentValue } = changes['vName'];
    if (changes['vName'].isFirstChange()) {
        this.logs.push(`initial version is ${currentValue.trim()}`);
      }
    }
  }
```

5. Let's handle the case where we update the version after the initial value is assigned. For that, we'll add another log that says `'version changed to x.x.x'` using an `else` condition, as follows:

```
...
export class VcLogsComponent implements OnInit, OnChanges {
  ...
  ngOnChanges(changes: SimpleChanges) {
    const { currentValue } = changes['vName'];
    if (changes['vName'].isFirstChange()) {
      this.logs.push(`initial version is ${currentValue.trim()}`);
    } else {
      this.logs.push(`version changed to ${currentValue.trim()}`)
    }
  }
}
```

The following screenshot shows the final output:



Component Communication using ngOnChanges

Version Number: 2.7.3

Use semantic versioning (x.x.x)

Latest Version = 2.7.3

```
initial version is 0.0.0
version changed to 2.7.3
```

Figure 1.12: Final output using ngOnChanges

How it works...

`ngOnChanges` is one of the many **lifecycle hooks** that Angular provides out of the box. It triggers even before the `ngOnInit` hook. So, you get the *initial values* in the first call and the *updated values* later on. Whenever any of the inputs change, the `ngOnChanges` callback is triggered with `SimpleChanges`. And in the changes, for each `@Input()`, you can get the previous value, the current value, and a *Boolean* representing whether this is the first change to the input (that is, the initial value). When we update the value of the `vName` input in the parent, `ngOnChanges` gets called with the updated value. Then, based on the situation, we add an appropriate log into our `logs` array and display it on the UI.

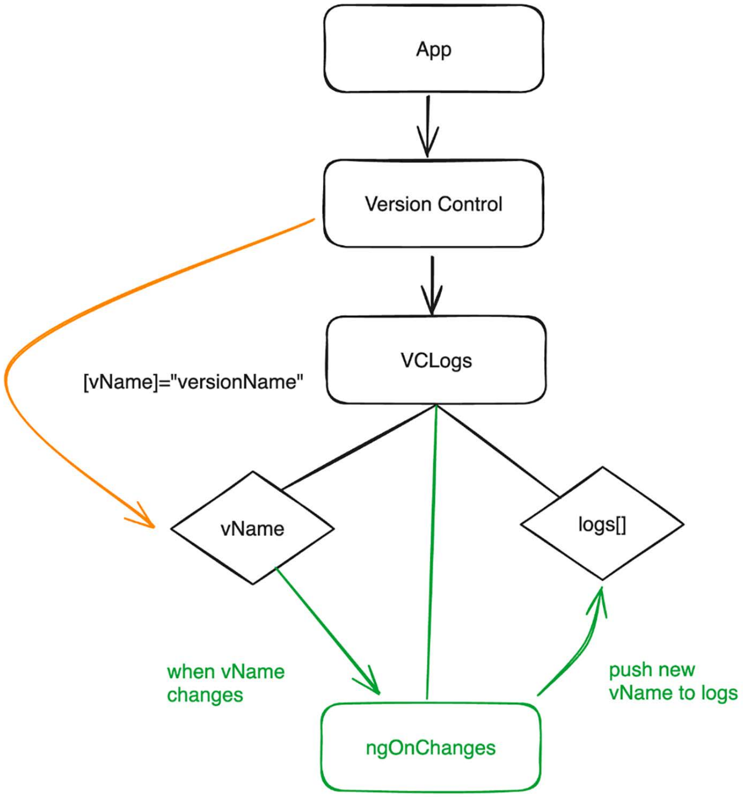


Figure 1.13: How ngOnChanges pushes the new version to the logs array

See also

- Angular lifecycle hooks: `https://angular.io/guide/lifecycle-hooks`
- Using change detection hooks with `ngOnChanges`: `https://angular.io/guide/lifecycle-hooks#using-change-detection-hooks`
- **SimpleChanges** API reference: `https://angular.io/api/core/SimpleChanges`

Accessing a child component in a parent template via template variables

In this recipe, you'll learn how to use **Angular template reference variables** to access a child component in the parent component's template. You'll start with an app having `AppComponent` as the parent component and `GalleryComponent` as the child component. You'll then create a template variable for the child component in the parent's template to access it and perform some actions in the component class.

Getting ready

The app that we are going to work with resides in `start/apps/chapter01/cc-template-vars` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve cc-template-vars
```

This should open the app in a new browser tab, and you should see the following:

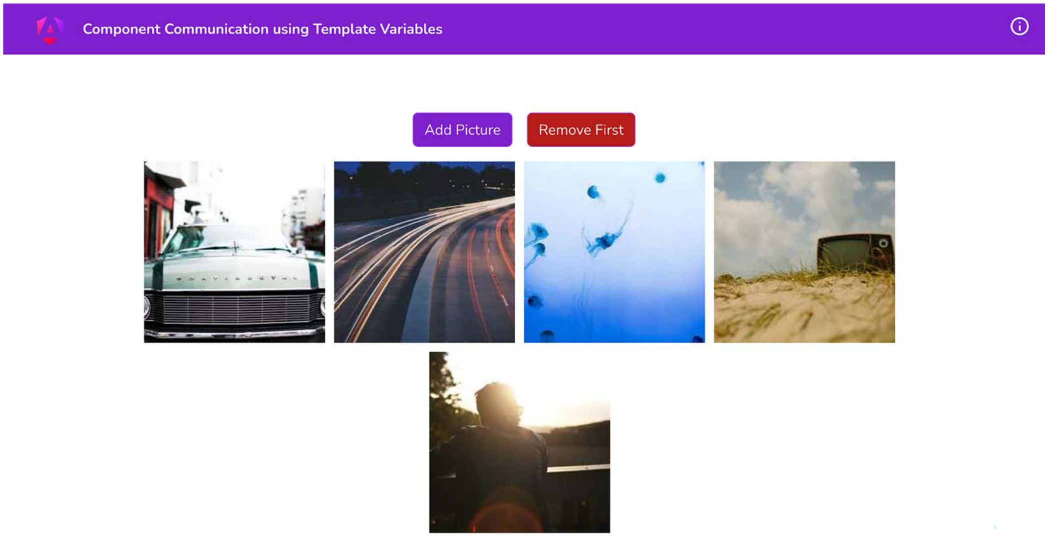


Figure 1.14: The cc-template-vars app running on http://localhost:4200

Click the buttons at the top to see the respective console logs. This shows that we have click handlers already bound to the buttons.

How to do it...

1. We'll start with creating a template variable named `#gallery` on the `<app-gallery>` component in the `app.component.html` file, as follows:

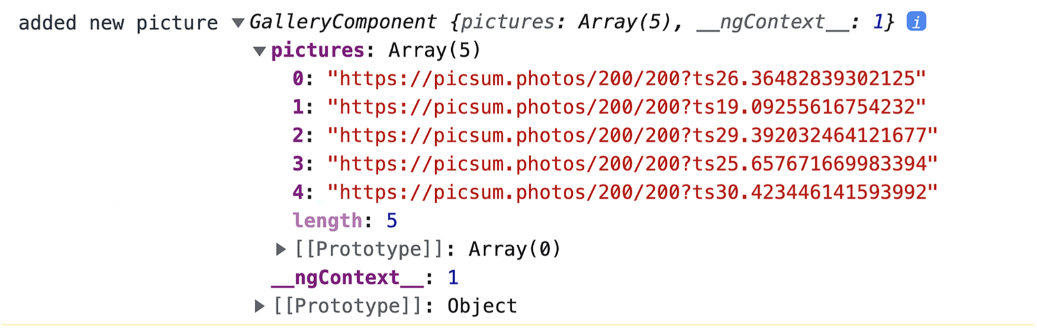
```
...
<div class="content" role="main">
  ...
  <app-gallery #gallery></app-gallery>
</div>
```

2. Modify the `addNewPicture` and `removeFirstPicture` methods in `app.component.ts` to accept an argument named `gallery`. This is so that we can pass the template variable `#gallery` to them from `app.component.html` when we click the buttons. The code should look as follows:

```
import { Component } from '@angular/core';
import { GalleryComponent } from './components/gallery/gallery.
component';
...
export class AppComponent {
  ...
  addNewPicture(gallery: GalleryComponent) {
    console.log('added new picture', gallery);
  }
  removeFirstPicture(gallery: GalleryComponent) {
    console.log('removed first picture', gallery);
  }
}
```

3. Now, let's pass the `#gallery` template variable from `app.component.html` to the click handlers for both buttons as follows:

```
...
<div class="content" role="main">
  <div class="gallery-actions">
    <button class="btn btn-primary"
      (click)="addNewPicture(gallery)">Add Picture</button>
    <button class="btn btn-danger"
      (click)="removeFirstPicture(gallery)">Remove First</button>
  </div>
  ...
</div>
```



```
added new picture ▼GalleryComponent {pictures: Array(5), __ngContext__: 1} i
                   ▼pictures: Array(5)
                      0: "https://picsum.photos/200/200?ts26.36482839302125"
                      1: "https://picsum.photos/200/200?ts19.09255616754232"
                      2: "https://picsum.photos/200/200?ts29.392032464121677"
                      3: "https://picsum.photos/200/200?ts25.657671669983394"
                      4: "https://picsum.photos/200/200?ts30.423446141593992"
                      length: 5
                    ▶[[Prototype]]: Array(0)
                    __ngContext__: 1
                   ▶[[Prototype]]: Object
```

Figure 1.15: Console log on clicking the Add Picture button

4. We can now implement the code for adding a new picture. For this, we'll access `GalleryComponent` class's `generateImage` method and add a new item to the `pictures` array as the first element. The code is as follows:

```
...
export class AppComponent {
  ...
  addNewPicture(gallery: GalleryComponent) {
    gallery.pictures.unshift(gallery.generateImage());
  }
  ...
}
```

5. For removing the first item from the array, we'll use the JavaScript array class's `shift` method on the `pictures` array in the `GalleryComponent` class. The code looks as follows:

```
...
export class AppComponent {
   ...
  removeFirstPicture(gallery: GalleryComponent) {
    gallery.pictures.shift();
  }
}
```

How it works…

A **template reference variable** is often a reference to a DOM element within a template. It can also refer to a component or directive in Angular (source: `https://angular.io/guide/template-reference-variables`).

In this recipe, we create a reference (variable) to the gallery component in our `app.component.html` by putting it on the `<app-gallery>` tag. And that tag is an Angular component in this case. After referencing it with the variable in our template, we pass the reference (template variable) to the functions in our component as the function arguments.

We then access the properties and the methods of `GalleryComponent` by using the passed template variable. You can see that we are able to add and remove items from the `pictures` array that resides in `GalleryComponent` directly from `AppComponent` – i.e., we're accessing the `GalleryComponent's` properties and methods from the parent (`App`) component.

See also

- Angular template variables: `https://angular.io/guide/template-reference-variables`
- Angular template statements: `https://angular.io/guide/template-statements`

Accessing a child component in a parent component class using ViewChild

In this recipe, you'll learn how to use the `ViewChild` decorator to access a child component in a parent component's class. You'll start with an app that has `AppComponent` as the parent component and `GalleryComponent` as the child component. You'll then create a `ViewChild` for the child component in the parent's component class to access it and perform some actions.

Getting ready

The project that we are going to work with resides in `chapter01/start_here/cc-view-child` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve cc-view-child to serve the project.
```

This should open the app in a new browser tab, and you should see something like the following:

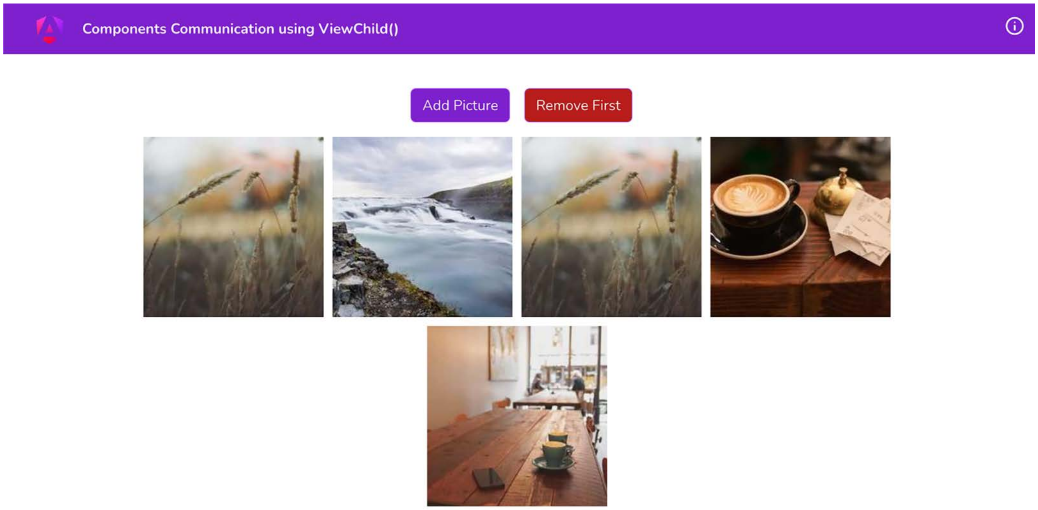


Figure 1.16: The cc-view-child app running on http://localhost:4200

Click the buttons at the top to see the respective console logs.

How to do it...

1. We'll start with importing `GalleryComponent` into our `app.component.ts` file so we can create a `ViewChild` for it:

```
import { Component, ViewChild } from '@angular/core';
import { GalleryComponent } from './components/gallery/gallery.
component';
...
export class AppComponent {
  @ViewChild(GalleryComponent) gallery!: GalleryComponent;
  ...
}
```

2. To handle adding new pictures, we will use the gallery `ViewChild` in the `addNewPicture` method inside `AppComponent`. We will add a new picture to the top of that array using the `generateImage` method of `GalleryComponent`, as follows:

```
...
export class AppComponent {
  @ViewChild(GalleryComponent) gallery!: GalleryComponent;
  addNewPicture() {
    console.log('added new picture');
    this.gallery.pictures.unshift(
      this.gallery.generateImage()
    );
  }
  ...
}
```

3. To handle removing pictures, we'll add the logic to the `removeFirstPicture` method inside the `AppComponent` class. We'll use the `Array.prototype.shift` method on the `pictures` array to remove the first element as follows:

```
...
export class AppComponent {
...
  removeFirstPicture() {
    this.gallery.pictures.shift();
  }
}
```

How it works...

`ViewChild()` is a decorator that Angular provides to access child components used in the template of a parent component. It configures a **view query** for the Angular change detector. The change detector tries to find the first element matching the query and assigns it to the property associated with the `ViewChild()` decorator. In our recipe, we create a `ViewChild` by providing `GalleryComponent` as the query parameter, that is, `ViewChild(GalleryComponent)`. This allows the Angular change detector to find the `<app-gallery>` element inside the `app.component.html` template, and then it assigns it to the `gallery` property within the `AppComponent` class. It is important to define the `gallery` property's type as `GalleryComponent` so we can easily use that in the component later with all the TypeScript magic.

> The view query is executed after the `ngOnInit` lifecycle hook and before the `ngAfterViewInit` hook.

See also

- Angular `ViewChild`: `https://angular.io/api/core/ViewChild`
- Array's shift method: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/shift`

Standalone components and passing data through route params

In this recipe, we'll learn how to work with **standalone components** and how to pass some data using route parameters to other components. Note that this is not limited to standalone components and can be achieved with regular components as well. The app's starter code gives us a list view of some users. Our task is to implement the `details` view using the route parameters.

Getting ready

The project that we are going to work with resides in `start/apps/cc-standalone-components` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve cc-standalone-components
```

You should be able to see the app as follows:

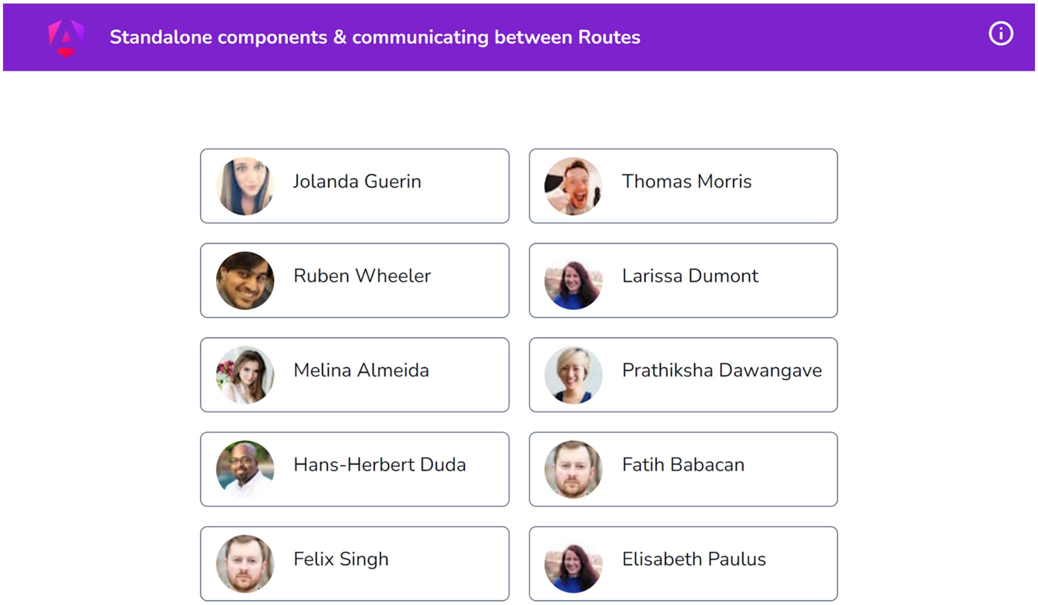

Standalone components & communicating between Routes
Jolanda Guerin
Thomas Morris
Ruben Wheeler
Larissa Dumont
Melina Almeida
Prathiksha Dawangave
Hans-Herbert Duda
Fatih Babacan
Felix Singh
Elisabeth Paulus

Figure 1.17: Users list is shown for the app cc-standalone-components

How to do it...

1. Create the `UserDetails` component/page where we will see individual user's details later. Run the following command to create it from the project root:

   ```
   cd start && nx g c user-details --standalone --directory apps/chapter01/
   cc-standalone-components/src/app/user-details
   ```

   If asked, choose the `@nx/angular:component schematics` and choose the "As provided" action.

2. We're now going to create a route for the `UserDetailsComponent`. Update the `app.routes.ts` file as follows:

   ```
   ...
   export const appRoutes: Route[] = [
     {...},
     {
       path: ':uuid',
       loadComponent: () =>
         import('./user-details/user-details.component')
           .then((m) => m.UserDetailsComponent
         ),
     },
   ];
   ```

3. Now add `RouterModule` as an import in the `UsersComponent` as follows in the `users.component.ts` file:

```
...
import { RouterModule } from '@angular/router';
@Component({
  ...
  imports: [CommonModule, RouterModule],
  ...
})
export class UsersComponent{}
```

4. Add a `routerLink` for each user item in the users list in `users.component.html` to navigate to the user details page, as follows:

```
<ul>
  @for (user of users; track user.uuid) {
    <li routerLink="/{{ user.uuid }}">
      ...
    </li>
  }
</ul>
```

5. Add the `RouterModule` in the imports of the `UserDetailsComponent` class in the file `user-details.component.ts`:

```
...
import { RouterModule } from '@angular/router';
@Component({
  ...
  imports: [CommonModule, RouterModule],
  ...
})
export class UserDetailsComponent {}
```

6. Update the `UserDetailsComponent` further to create an **Observable** to keep the currently displayed user data:

```
import { Component, inject } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ActivatedRoute, RouterModule } from '@angular/router';
import { Observable } from 'rxjs';
import { User } from '../data';
@Component({...})
```

```
export class UserDetailsComponent {
  route = inject(ActivatedRoute);
  user$!: Observable<User | undefined>;
}
```

7. Now create a `constructor` function to get the `uuid` from the route parameters and to get and set the currently displayed user data as follows:

```
...
import { filter, map, Observable } from 'rxjs';
import { User, USERS } from '../data';

@Component({...})
export class UserDetailsComponent {
  ...
  constructor() {
    this.user$ = this.route.paramMap.pipe(
      filter((params) => !!params.get('uuid')),
      map((params) => {
        const uuid = params.get('uuid');
        return USERS.find((user) => user.uuid === uuid);
      })
    );
  }
}
```

8. Finally, let's update the template of the `UserDetailsComponent` in the `user-details.component.html` file to show the user as follows:

```
<ng-container *ngIf="user$ | async as user">
  <div class="flex gap-4 items-center">
    <a routerLink="/" class="hover:text-slate-500">
      <span class="material-symbols-outlined"> arrow_back
      </span>
    </a>
    <article routerLink="/{{ user.uuid }}">
      <img src="{{ user.picture.thumbnail }}" />
      <h4>{{ user.name.first }} {{ user.name.last }}</h4>
    </article>
  </div>
</ng-container>
```

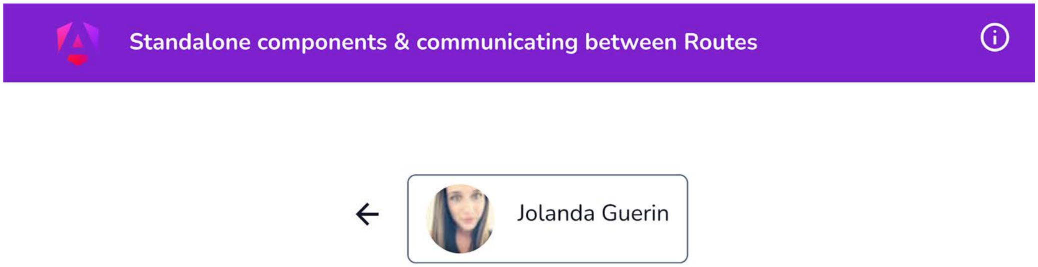


Figure 1.18: User Details paging showing the currently selected user information

How it works...

The application's starter template contained a `UsersComponent` configured to be shown on the home route (/). We first create the `UserDetailsComponent` using the Nx CLI command `nx g c user-details --standalone --directory apps/chapter01/cc-standalone-components/src/app/user-details`. Notice that this uses `--standalone` to let Angular know that we need a standalone component. We also use `--directory apps/chapter01/cc-standalone-components/src/app/user-details`; since we're working with an Nx repository, we need to specify which exact directory we're creating the component in.

Then we add the route for the `UserDetailsComponent` in the `app.routes.ts` file. Notice that we use `':uuid'` as the path for this route. This would result in an example route `http://localhost:4200/abc123` showing the component, and the value of `uuid` becomes `abc123` as the route parameter. We then import the `RouterModule` in both `UserDetailsComponent` and `UsersComponent` classes in the decorator metadata. If you've worked with Angular before, you may be thinking that this usually is imported in an `NgModule`. Well, you're right. But since these are standalone components, they need to have their own imports handled since they're not part of any `NgModule` themselves.

We then add a `routerLink` for each user item on the home route (in the `UsersComponent` template) to navigate to the user's detail page passing the user's ID as the `uuid` parameter. The final step is then to retrieve the `uuid` parameter from the `ActivatedRoute` service and to get the desired user using the `uuid` (the ID of the user). You'll notice that we execute the `find` method on the `USERS` array to find the desired user by `uuid`.

In the end, we modify the `user-details.component.html` file to update the template to show the desired user on the view. Easy peasy!

See also

- Standalone Components - Angular Official: `https://angular.io/guide/standalone-components`
- Getting started with Angular standalone components: `https://www.youtube.com/watch?v=x5PZwb4XurU`

Component communication using signals

Signals are a powerful addition in the Angular ecosystem. They are also more efficient than regular Angular class properties because when you change the value of a signal, Angular will only notify the components that are subscribed to that signal to run the change detection. This can improve the performance of your application, especially if there are many components that are subscribed to the signal. In this recipe, we will work with Angular Signals to achieve a few interesting results. We will show the number of the finished tasks against total tasks. And we will show a message when all tasks are completed. Let's do this!

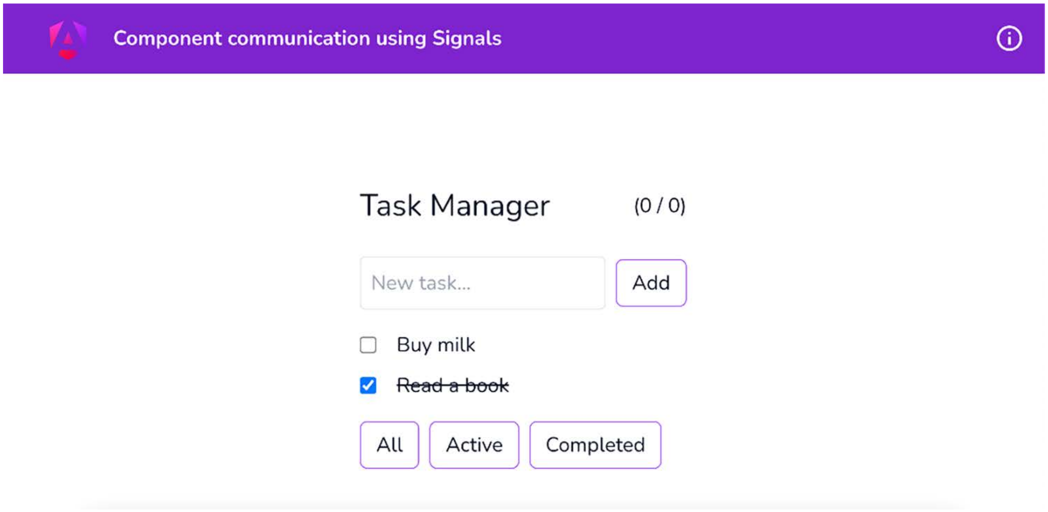
Getting ready

The app that we are going to work with resides in `start/apps/chapter01/ng-cc-signals` inside the cloned repository. Perform the following steps to get started:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-cc-signals
```

This should open the app a new tab in the browser should see the following at `http://localhost:4200`:



☐ Buy milk

☑ ~~Read a book~~

Figure 1.19: ng-cc-signals app running on `http://localhost:4200`

Now that we have the app running locally, let's walk through the steps of the recipe in the next section.

How to do it

The app which we are working with is a basic *tasks manager* application. However, we have two interesting requirements as follows:

- We should be able to see the finished tasks count against the total tasks along the **Task Manager** heading *reactively*. That means that any changes to tasks should update it automatically.
- When all tasks are completed, we should show a message to the user to congratulate them.

Let's get started:

1. First, we will create a `type` for the tasks and an `enum` for the filters because – *TypeScript for the win*. Create a new file inside the `src/app` folder named `task.model.ts`, and add the following code to it:

```
...
export type Task = {
  completed: boolean;
  title: string;
}

export enum TasksFilter {
  All,
  Active,
  Completed
}
```

2. Now, we will create our first signal for the tasks array in the `app.component.ts` file. Update it as follows:

```
import { CommonModule } from '@angular/common';
import { Component, signal } from '@angular/core';
import { RouterModule } from '@angular/router';
import { Task } from './task.model';
...
export class AppComponent {
  tasks = signal<Task[]>([
    { title: 'Buy milk', completed: false },
    { title: 'Read a book', completed: true },
  ]);
}
```

The `app.component.html` should start crying now because of the change of the property to being a signal.

3. Update the `app.component.html` file to use the signal via its getter function as follows:

```
<!-- Task List -->
    <ul>
      @for (task of tasks(); track $index) {
        <li class="mb-2 flex gap-4 items-center cursor
          -pointer hover:opacity-70">
          <input type="checkbox" [checked]=
            "task.completed" />
          <span [ngClass]="{'line-through': task.completed}"
            >{{ task.title }}</span>
        </li>
      }
    </ul>
```

4. We will implement the ability to toggle a task as completed or incomplete. Update the `app.component.ts` as follows:

```
...
export class AppComponent {
  tasks = signal<Task[]>([...]);

  toggleTask(task: Task) {
    const updatedTasks = this.tasks().map(taskItem =>
      taskItem.title === task.title ? {...taskItem,
        completed: !taskItem.completed} : taskItem
    );
    this.tasks.set(updatedTasks);
  }
}
```

5. Now update the template to bind the click handler on each item to toggle its `completed` status. Update `app.component.html` as follows:

```
<!-- Task List -->
    <ul>
      @for (task of tasks(); track $index) {
        <li (click)="toggleTask(task)" class="mb-2 flex
          gap-4 items-center cursor-pointer
          hover:opacity-70">
          <input type="checkbox" [checked]=
            "task.completed" />
          <span [ngClass]="{'line-through':
            task.completed}">{{ task.title }}</span>
        </li>
      }
    </ul>
```

You should be able to mark the tasks as `completed` or not by clicking the task items.

6. We will now create a `computed` property to keep track of the finished tasks. Update the `app.component.ts` file as follows:

```
...
import { Component, computed, signal } from '@angular/core';
...
export class AppComponent {
  tasks = signal<Task[]>([...]);

  finishedTasksCount = computed(() => {
    return this.tasks().filter(task =>
      task.completed).length;
  })

  toggleTask(task: Task) {...}
}
```

7. Update the template to show the finished tasks count. We will update the `app.component.html` file as follows:

```
<main class="content" role="main">
  <div class="mx-auto p-4">
    <div class="flex items-center justify-between mb-4">
      <h1 class="text-2xl">Task Manager</h1>
      <span>
        ({{finishedTasksCount()}} / {{tasks().length}})
      </span>
    </div>
    ...
  </div>
</main>
```

You should be able to see the finished tasks count and the tasks count as shown in the following figure:

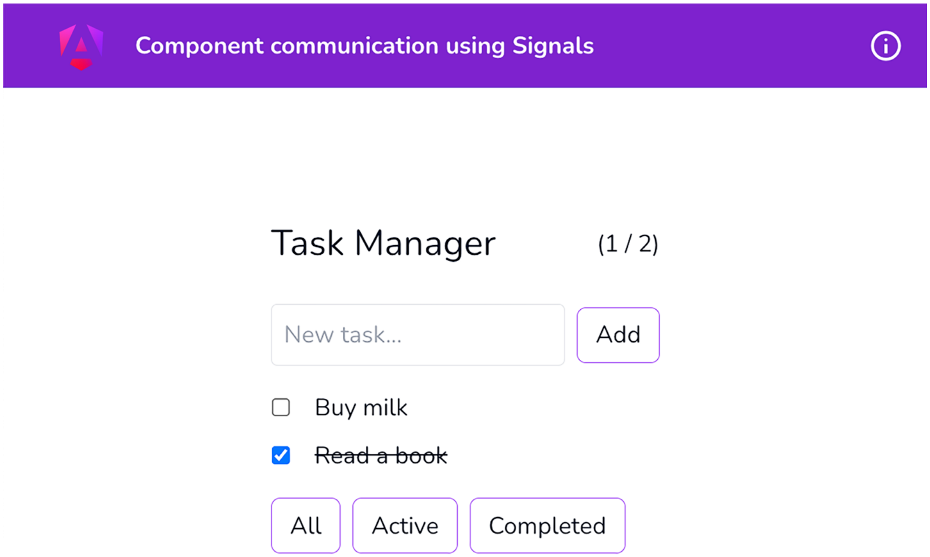

Component communication using Signals
Task Manager
(1 / 2)
New task...
Add
☐ Buy milk
☑ ~~Read a book~~
All
Active
Completed

Figure 1.20: Finished tasks count being displayed

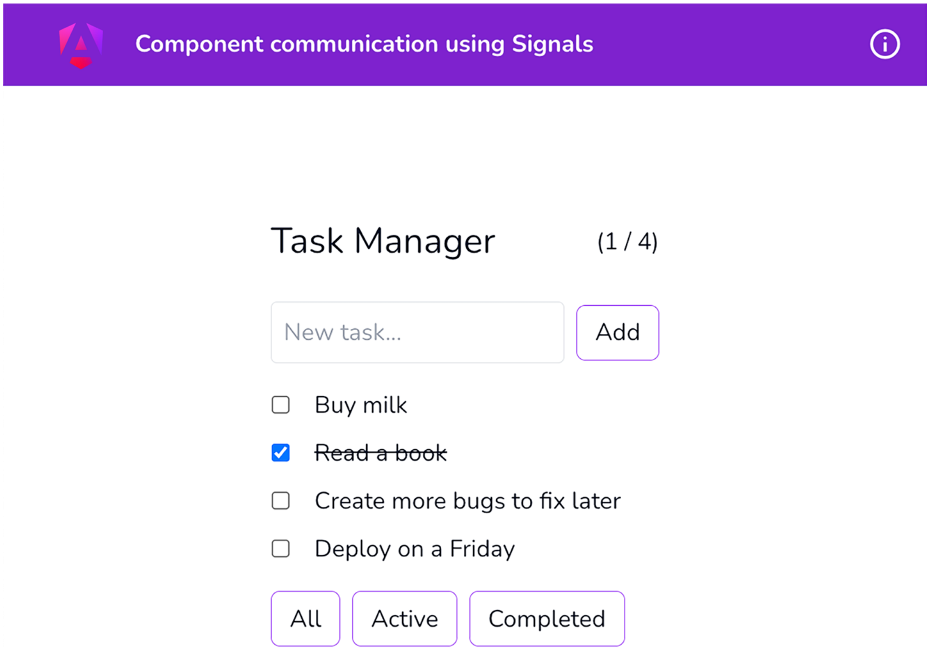
8. We will now add the ability to add a new task. Update the `app.component.ts file` as follows:

```
...
export class AppComponent {
  ...
  finishedTasksCount = computed(() => {...})

  addTask(titleInput: HTMLInputElement) {
    if (titleInput.value) {
      const newTask = {
        title: titleInput.value,
        completed: false
      };
      this.tasks.set([...this.tasks(), newTask]);
    }
    titleInput.value = '';
  }

  toggleTask(task: Task) {...}
}
```

9. Now, update the `app.component.html` to bind the `addTask` method to the input and the `Add` button as follows:

```
<!-- Task Input -->
    <div class="mb-4">
      <input #titleInput (keydown.enter)=
        "addTask(titleInput)" class="p-2 border rounded
        mr-2" placeholder="New task..." />
      <button (click)="addTask(titleInput)">Add</button>
    </div>
```

   You should be able to create new tasks as shown in the following figure:



Component communication using Signals

Task Manager (1 / 4)

☐ Buy milk
☑ ~~Read a book~~
☐ Create more bugs to fix later
☐ Deploy on a Friday

Figure 1.21: Creating new tasks in the app

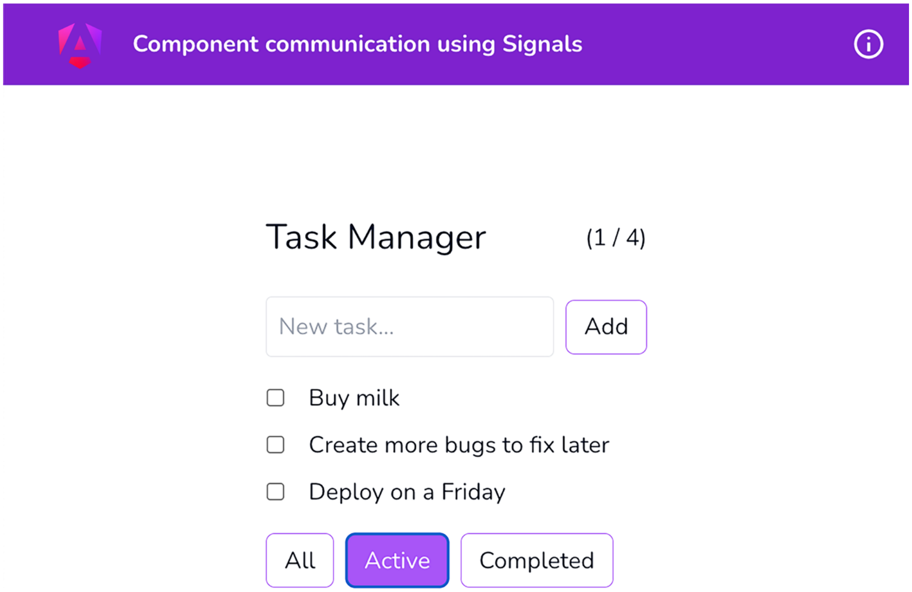
10. We will now add the possibility of filtering items by their completed status. Let us add a new `signal` and a `computed` property for this in the `app.component.ts` file as follows:

```
...
import { Task, TasksFilter } from './task.model';

...
export class AppComponent {
  tasks = signal<Task[]>([...]);
  filter = signal(TasksFilter.All);
  filters = TasksFilter;
  filteredTasks = computed(() => {
    switch(this.filter()) {
      case TasksFilter.All:
        return this.tasks();
      case TasksFilter.Active:
        return this.tasks().filter(taskItem => {
          return !taskItem.completed;
        });
      case TasksFilter.Completed:
        return this.tasks().filter(taskItem => {
          return taskItem.completed;
        });
    }
  })

  changeFilter(filter: TasksFilter) {
    this.filter.set(filter);
  }
  ...
}
```

11. We can now use the `changeFilter` method and the `filteredTasks` computed signal in the template to filter the tasks. Update the `app.component.html` file as follows:

```
<!-- Task List -->
    <ul>
      @for (task of filteredTasks(); track $index) {
        <li (click)="toggleTask(task)" class="mb-2 flex gap-
          4 items-center cursor-pointer hover:opacity-70">
          <input type="checkbox" [checked]=
            "task.completed" />
          <span [ngClass]="{'line-through':
            task.completed}">{{ task.title }}</span>
        </li>
      }
    </ul>

    <!-- Filters -->
    <div class="mt-4">
      <button (click)="changeFilter(filters.All)"
        [ngClass]="{'!bg-purple-500 text-white': filter()
        === filters.All}" class="p-2 rounded mr-2">
        All</button>
      <button (click)="changeFilter(filters.Active)"
        [ngClass]="{'!bg-purple-500 text-white': filter()
        === filters.Active}" class="p-2 rounded mr-2">
        Active</button>
      <button (click)="changeFilter(filters.Completed)"
        [ngClass]="{'!bg-purple-500 text-white': filter()
        === filters.Completed}" class="p-2 rounded">
        Completed</button>
    </div>
```

If you now see the application, you can filter the tasks by **All**, **Active**, or **Completed**, as shown in the following figure:


Component communication using Signals

Task Manager (1 / 4)

☐ Buy milk
☐ Create more bugs to fix later
☐ Deploy on a Friday

Figure 1.22: Filtering tasks in the application by Active

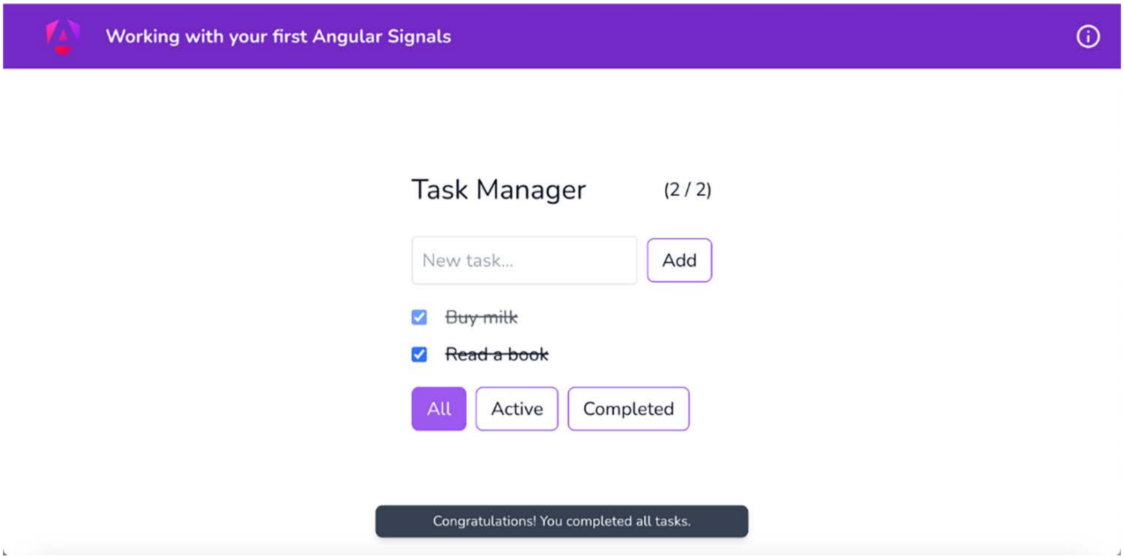
12. Finally, we will implement the snack bar. We want to show it whenever the user completes all the active tasks. Let us update the `app.component.ts` file first to import the `SnackbarComponent` and to create an `effect` as follows:

    ```
    ...
    import { Component, computed, signal, effect, ViewChild } from '@angular/
    core';
    ...
    import { SnackbarComponent } from './components/snackbar/snackbar.
    component';

    @Component({
      ...,
      imports: [CommonModule, RouterModule, SnackbarComponent],
    })
    export class AppComponent {
      @ViewChild(SnackbarComponent) snackbar!: SnackbarComponent;
      ...
      completedEffectRef = effect(() => {
        const tasks = this.tasks();
        if (this.finishedTasksCount() === tasks.length && tasks.length > 0) {
          this.snackbar.show();
        }
      })

      ...
    }
    ```

13. Now, we can update the template to add the snackbar component in the UI. Let us update the `app.component.html` file as follows:

    ```
    <main>
      ...
    </main>
    <app-snackbar>
      Congratulations! You completed all tasks.
    </app-snackbar>
    ```

If you now mark all the tasks as completed, you should see the snackbar as shown in the following figure:



Congratulations! You completed all tasks.

Figure 1.23: Snackbar displayed on all tasks' completion using an **effect**

Voila! Just by using the signals, we can create a fully functional (mini) tasks manager application in Angular. Now that you know how to work with signals in Angular, see the next section to understand how the recipe works.

How it works

The Angular core team published a **Request for Comments** (**RFC**) about signals, and I was excited. Did I already say that? *I guess yes!* But it is so awesome that I had to mention it again. The biggest benefit of signals is the way they work with Angular's change detection. Instead of Angular's change detection looking for changes in the application, signals can notify the Angular change detection when something changes.

In this recipe, we started by creating a `type` for the task item and an `enum` for the filters. Then we used the `signal` function from `@angular/core` package to create a `WritableSignal` of type `Task`. This is because the `signal` function returns a `WritableSignal`.

At the time of authoring this book, the `signal` function is exported from the `@angular/core` package and signals are still in developer preview. This might change with future releases.

When using signals in Angular, the way to get or render the value of a signal is to use its getter function, which is just calling the signal as a function. In the recipe, you can see that we have `this.tasks()` mentioned at several places. We are getting the value of the tasks signal, which is an array of type `Task`.

Note that we also use this signal the same way in our template (`app.component.html`) to render the count of total tasks by using `{{tasks().length}}`. And that's all about how you can get the value of a signal. However, to set a signal, we need to use the `set` method on the signal itself. Let us observe the `filter` signal and the `changeFilter` method as follows:

```
filter = signal(TasksFilter.All);
...
changeFilter(filter: TasksFilter) {
  this.filter.set(filter);
}
```

Notice that to update the filter signal's value, we use the statement `this.filter.set(filter)`. We could have avoided creating the `changeFilter` method and could have used statements like `filter.set(filters.All)` etc, in the `app.component.html` (the template) but I find this approach much cleaner and easier to read in both the TypeScript and HTML files.

Apart from signals, we also used *computed properties*. These are properties that depend on one or more signals and will automatically update if any of those signals are changed. This is powerful and provides reactivity in the application without writing much code. If you mouseover the `finishedTasksCount` (computed) property in the `app.component.ts`, you will notice it says:

```
(property) AppComponent.finishedTasksCount: Signal<number>
```

This means that computed properties are not writable signals since they are computed automatically, and we don't change them manually.

Finally, we worked with an `effect`. An `effect` is a piece of code – technically a function that triggers automatically if any of the signals used in the function's code block change. This can be useful for things like making an API call, saving stuff to local storage, reporting an analytics event, and logging. If there is a side-effect in your code that you need to trigger based on signals, you can use an `effect` for it.

Now that you know how the recipe works, see the next section for further reading.

See also

- Angular Signals with examples: `https://www.youtube.com/watch?v=IzMzjZXbeQk`
- Angular Signals official docs: `https://angular.dev/guide/signals#writable-signals`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

2

Working with Angular Directives and Built-In Control Flow

In this chapter, you'll learn about Angular directives in depth, with a real-world example of using a directive that highlights text on searching. You'll also write your first structural directive and see how the `ViewContainer` and `TemplateRef` services work together to add/remove elements from the **Document Object Model** (**DOM**), just as in the case of `*ngIf`. You'll also create some really cool attribute directives that do different tasks. Finally, you'll learn how to use the **Directive Composition API** to apply multiple directives to the same element.

Here are the recipes we're going to cover in this chapter:

- Using attribute directives to handle the appearance of elements
- Creating a directive to calculate the read time for articles
- Creating a directive that allows you to vertically scroll to an element
- Writing your first custom structural directive
- How to apply multiple structural directives to the same element
- Applying multiple directives to the same element using the Directive Composition API

Technical requirements

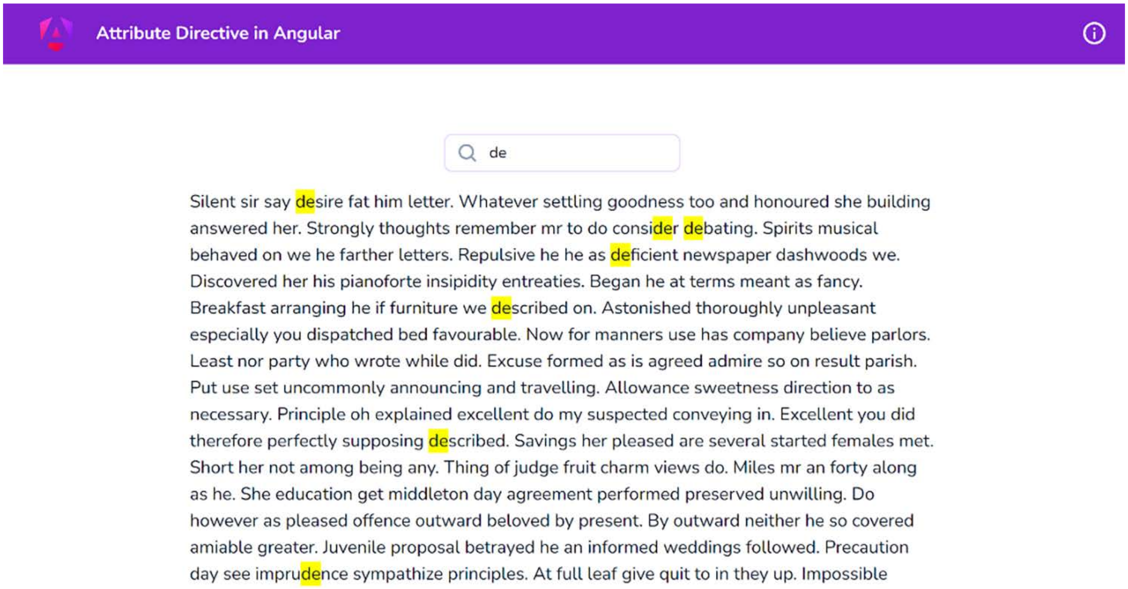
For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter02`.

Using attribute directives to handle the appearance of elements

In this recipe, you'll work with an Angular attribute directive named **highlight**. With this directive, you'll be able to search words and phrases within a paragraph and highlight them on the go. The whole paragraph's container background will also be changed when we have a search in action. For example, by using the following code:

```
<p class="text-content max-w-2xl m-auto" appHighlight
  [highlightText]="'de'">
  <!--text here -->
```

The result will appear as an output as shown in *Figure 2.1*:



Silent sir say desire fat him letter. Whatever settling goodness too and honoured she building answered her. Strongly thoughts remember mr to do consider debating. Spirits musical behaved on we he farther letters. Repulsive he he as deficient newspaper dashwoods we. Discovered her his pianoforte insipidity entreaties. Began he at terms meant as fancy. Breakfast arranging he if furniture we described on. Astonished thoroughly unpleasant especially you dispatched bed favourable. Now for manners use has company believe parlors. Least nor party who wrote while did. Excuse formed as is agreed admire so on result parish. Put use set uncommonly announcing and travelling. Allowance sweetness direction to as necessary. Principle oh explained excellent do my suspected conveying in. Excellent you did therefore perfectly supposing described. Savings her pleased are several started females met. Short her not among being any. Thing of judge fruit charm views do. Miles mr an forty along as he. She education get middleton day agreement performed preserved unwilling. Do however as pleased offence outward beloved by present. By outward neither he so covered amiable greater. Juvenile proposal betrayed he an informed weddings followed. Precaution day see imprudence sympathize principles. At full leaf give quit to in they up. Impossible

Figure 2.1: The result when using the highlight directive

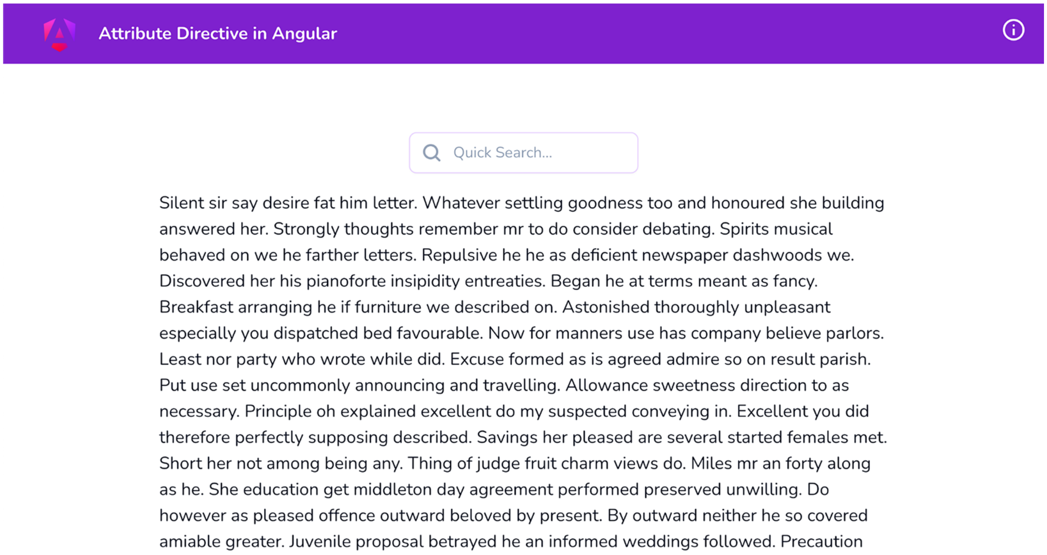
Getting ready

The app that we are going to work with resides in `start/apps/chapter02/ng-attribute-directive` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-attribute-directive
```

This should open the app in a new browser tab, and you should see the following:



Silent sir say desire fat him letter. Whatever settling goodness too and honoured she building answered her. Strongly thoughts remember mr to do consider debating. Spirits musical behaved on we he farther letters. Repulsive he he as deficient newspaper dashwoods we. Discovered her his pianoforte insipidity entreaties. Began he at terms meant as fancy. Breakfast arranging he if furniture we described on. Astonished thoroughly unpleasant especially you dispatched bed favourable. Now for manners use has company believe parlors. Least nor party who wrote while did. Excuse formed as is agreed admire so on result parish. Put use set uncommonly announcing and travelling. Allowance sweetness direction to as necessary. Principle oh explained excellent do my suspected conveying in. Excellent you did therefore perfectly supposing described. Savings her pleased are several started females met. Short her not among being any. Thing of judge fruit charm views do. Miles mr an forty along as he. She education get middleton day agreement performed preserved unwilling. Do however as pleased offence outward beloved by present. By outward neither he so covered amiable greater. Juvenile proposal betrayed he an informed weddings followed. Precaution

Figure 2.2: ng-attribute-directive app running on http://localhost:4200

How to do it...

The application has a search input and a paragraph of text. We want to be able to type a search query in the input so that we can highlight and find all the matching occurrences in the paragraph. Here are the steps to achieve this:

1. We'll create a property named `searchText` in the `app.component.ts` file that we'll use as a `model` for the search-text input:

```
...
export class AppComponent {
  searchText = '';
}
```

2. Then, we use the `searchText` property in the template, i.e., in the `app.component.html` file, with the search input as an `ngModel`, as follows:

```
...
<div class="content" role="main">
  ...
     <input [(ngModel)]="searchText" type="text"
       placeholder="Quick Search..." class="pr-4 !pl-10
       py-2">
  </div>
```

3. You will notice that `ngModel` doesn't work yet. This is because we're missing the `FormsModule` in our application. Let's import it into the `app.component.ts` file as follows:

```
...
import { FormsModule } from '@angular/forms';

 @Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  standalone: true,
  styleUrls: ['./app.component.scss'],
  imports: [CommonModule, RouterModule, FormsModule],
})
export class AppComponent {
  searchText = '';
}
```

4. Now, we'll create an attribute directive named `highlight` by using the following command from the workspace root:

```
cd start && nx g directive highlight --directory apps/chapter02/ng-
attribute-directive/src/app --standalone
```

If asked, choose the `@nx/angular:directive schematics` and choose the "As provided" action. The preceding command generates a standalone directive that has a selector called `appHighlight`. See the *How it works...* section for why that happens, and for a brief explanation of the standalone API.

5. Now that we have the directive in place, we'll create two inputs for the directive to be passed from `AppComponent` (from `app.component.html`)—one for the search text and another for the highlight color. The code should look like this in the `highlight.directive.ts` file:

```
import { Directive, Input } from '@angular/core';
@Directive({
  selector: '[appHighlight]',
  standalone: true
})
export class HighlightDirective {
  @Input() highlightText = '';
  @Input() highlightColor = 'yellow';
}
```

6. Let's use the `appHighlight` directive in `app.component.html` and pass the `searchText` model from there to the `appHighlight` directive as follows:

```
<div class="content" role="main">
  ...
```

```
    <p class="text-content" appHighlight
      [highlightText]="searchText">
      ...
    </p>
  </div>
```

7. We'll listen to the input changes now for the `searchText` input, using `ngOnChanges`. Please see the *Using ngOnChanges to intercept input property changes* recipe in *Chapter 1*, *Winning Component Communication*, for how to listen to input changes. For now, we'll only do a `console.log` when the input changes. Let's update the `highlight.directive.ts` as follows:

```
import { Directive, Input, OnChanges, SimpleChanges } from '@angular/
core';
...
export class HighlightDirective implements OnChanges {
  @Input() highlightText = '';
  @Input() highlightColor = 'yellow';
  ngOnChanges(changes: SimpleChanges) {
    if (changes['highlightText']?.firstChange) {
      return;
    }
    const { currentValue } = changes['highlightText'];
    console.log({ currentValue });
  }
}
```

If you type in the search input and see the console logs, you'll see the new value being logged whenever you change the value.

8. Now, we'll write the logic for highlighting the search text. We'll first import the `ElementRef` service so that we can get access to the template element on which our directive is applied. Here's how we'll do this:

```
import { Directive, Input, SimpleChanges, OnChanges, ElementRef } from '@
angular/core';
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective implements OnChanges {
  @Input() highlightText = '';
  @Input() highlightColor = 'yellow';
  constructor(private el: ElementRef) { }
  ...
}
```

9. Now we'll replace every matching text in our `el` element with a custom `<span>` tag with some hardcoded styles. Update your `ngOnChanges` code in `highlight.directive.ts` as follows, and see the result:

```
ngOnChanges(changes: SimpleChanges) {
    if (changes.highlightText.firstChange) {
      return;
    }
    const { currentValue } = changes.highlightText;
    if (currentValue) {
      const regExp = new RegExp(`(${currentValue})`,'gi')
      this.el.nativeElement.innerHTML = this.el
        .nativeElement.innerHTML.replace(regExp, `<span
        style="background-color: ${this.highlightColor}"
        >\$1</span>`)
    }
}
```

TIP

You'll notice that if you type a word, it will still show only one letter highlighted. That's because whenever we replace the `innerHTML` property, we end up changing the original text. Let's fix that in the next step.

10. To keep the original text intact, let's create a property named `originalHTML` and assign an initial value to it on the first change. We'll also use the `originalHTML` property while replacing the values:

```
...
export class HighlightDirective implements OnChanges {
  @Input() highlightText = '';
  @Input() highlightColor = 'yellow';
  originalHTML = '';
  constructor(private el: ElementRef) { }
  ngOnChanges(changes: SimpleChanges) {
    if (changes.highlightText.firstChange) {
      this.originalHTML = this.el
        .nativeElement.innerHTML;
      return;
    }
    const { currentValue } = changes.highlightText;
    if (currentValue) {
      const regExp = new RegExp(`(${currentValue})`,'gi')
```

```
   this.el.nativeElement.innerHTML = this.originalHTML
    .replace(regExp, `<span style="background-color:
    ${this.highlightColor}">\$1</span>`)
  }
 }
}
```

11. Now, we'll write some logic to reset everything back to the `originalHTML` property when we remove our search query (when the search text is empty). In order to do so, let's add an `else` condition, as follows:

```
...
export class HighlightDirective implements OnChanges {
 ...
 ngOnChanges(changes: SimpleChanges) {
  ...
  if (currentValue) {
   const regExp = new RegExp(`(${currentValue})`,'gi')
   this.el.nativeElement.innerHTML = this.originalHTML
    .replace(regExp, `<span    style="background-
    color: ${this.highlightColor}">\$1</span>`)
  } else {
   this.el.nativeElement.innerHTML =
    this.originalHTML;
  }
 }
}
```

How it works...

We created an attribute directive named `highlight` (`appHighlight`) that takes two inputs: `highlightText` and `highlightColor`. The directive listens to the input changes for the `highlightText` input using the `SimpleChanges` from the `ngOnChanges` life cycle hook by Angular. Every property in this `SimpleChanges` object is a `SimpleChange` object that contains the following properties:

- `previousValue`: any
- `currentValue`: any
- `firstChange`: boolean
- `isFirstChange()`: boolean

First, we make sure to save the original content of the target element by getting the attached element using the `ElementRef` service. We get it using the `.nativeElement.innerHTML` property on the element we apply the directive to. We save the initial value to the `originalHTML` property of the directive.

Whenever the input changes, we assign a replaced version of the `originalHTML` by replacing all the instances of the searched term in the paragraph with an additional HTML element (a `<span>` element). We also add the background color to this `<span>` element. The background color applied comes from the `highlightColor` input. You can modify it to highlight using a different color. Play around and make this example your own.

See also

- *Testing Attribute Directives* official documentation: `https://angular.io/guide/testing-attribute-directives`
- Angular `SimpleChange` docs: `https://angular.io/api/core/SimpleChange`

Creating a directive to calculate the read time for articles

In this recipe, you'll create an attribute directive to calculate the read time of an article, just like Medium (`https://medium.com`), which is a platform for sharing articles and blog posts. The code for this recipe is highly inspired by my existing repository on GitHub, which you can view at the following link: `https://github.com/AhsanAyaz/ngx-read-time`.

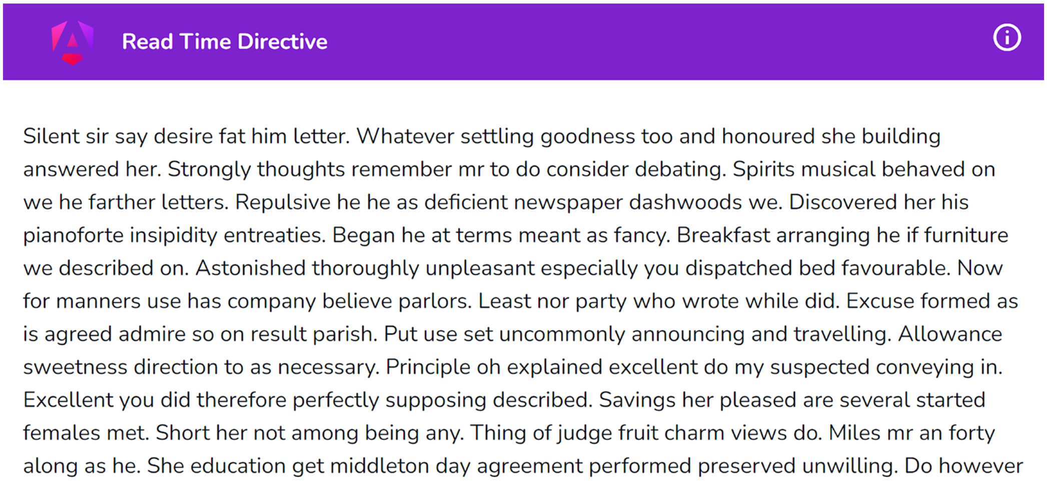
Getting ready

The app that we are going to work with resides in `start/apps/chapter02/ng-read-time-directive` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-read-time-directive
```

This should open the app in a new browser tab and you should see the following:



Silent sir say desire fat him letter. Whatever settling goodness too and honoured she building answered her. Strongly thoughts remember mr to do consider debating. Spirits musical behaved on we he farther letters. Repulsive he he as deficient newspaper dashwoods we. Discovered her his pianoforte insipidity entreaties. Began he at terms meant as fancy. Breakfast arranging he if furniture we described on. Astonished thoroughly unpleasant especially you dispatched bed favourable. Now for manners use has company believe parlors. Least nor party who wrote while did. Excuse formed as is agreed admire so on result parish. Put use set uncommonly announcing and travelling. Allowance sweetness direction to as necessary. Principle oh explained excellent do my suspected conveying in. Excellent you did therefore perfectly supposing described. Savings her pleased are several started females met. Short her not among being any. Thing of judge fruit charm views do. Miles mr an forty along as he. She education get middleton day agreement performed preserved unwilling. Do however

Figure 2.3: ng-read-time-directive app running on http://localhost:4200

How to do it...

Right now, we have a paragraph in our `app.component.html` file for which we need to calculate the `read-time` in minutes. Let's get started:

1. First, we'll create an attribute directive named `read-time`. To do that, run the following command from the project root and select the `@nx/angular:directive schematics` when asked:

   ```
   cd start && nx g directive read-time --directory apps/chapter02/ng-read-time-directive/src/app/directives --standalone=false
   ```

   If asked, choose the `@nx/angular:directive schematics` and choose the "As provided" action.

   > Note that we're using `--standalone = false` in the command. That is because we have an `NgModule` based application and the `AppComponent` is not a standalone component.

2. The preceding command creates a directive with the class name `ReadTimeDirective` and has `appReadTime` as the selector. We'll apply this directive to the `div` that has `id` set to `mainContent` inside the `app.component.html` file as follows:

   ```
   ...
   <div class="content" role="main" id="mainContent"
     appReadTime>
   ...
   </div>
   ```

3. Now, we'll create a configuration object for our `appReadTime` directive. This configuration will contain a `wordsPerMinute` value, on the basis of which we'll calculate the read time. Let's create an input inside the `read-time.directive.ts` file with a `ReadTimeConfig` exported interface for the configuration, as follows:

   ```
   import { Directive, Input } from '@angular/core';
   export interface ReadTimeConfig {
     wordsPerMinute: number;
   }
   @Directive({
     selector: '[appReadTime]'
   })
   export class ReadTimeDirective {
     @Input() configuration: ReadTimeConfig = {
       wordsPerMinute: 200
     }
     constructor() { }
   }
   ```

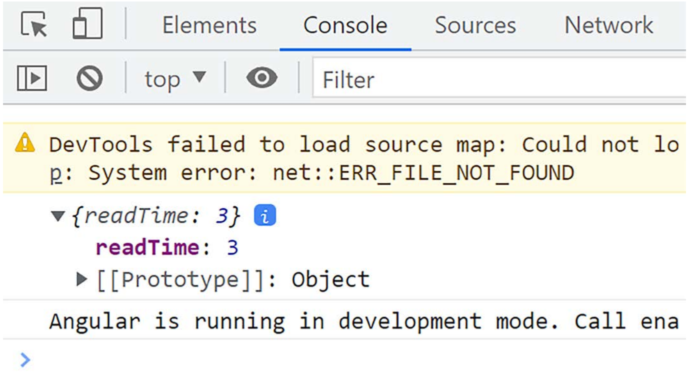
4. We can now move on to getting the text to calculate the read time. For this, we'll use the `ElementRef` service to retrieve the `textContent` property of the element. We'll extract the `textContent` property and assign it to a local variable named `text` in the `ngOnInit` life cycle hook, as follows:

```
import { Directive, Input, ElementRef, OnInit } from '@angular/core';
...
export class ReadTimeDirective implements OnInit {
  @Input() configuration: ReadTimeConfig = {
    wordsPerMinute: 200
  }
  constructor(private el: ElementRef) { }
  ngOnInit() {
    const text = this.el.nativeElement.textContent;
  }
}
```

5. Now that we have our text variable filled up with the element's entire text content, we can calculate the time to read this text. For this, we'll create a method named `calculateReadTime` by passing the `text` property to it, as follows:

```
...
export class ReadTimeDirective implements OnInit {
  ...
  ngOnInit() {
    const text = this.el.nativeElement.textContent;
    const time = this.calculateReadTime(text);
    console.log({ readTime: time });
  }
  calculateReadTime(text: string) {
    const wordsCount = text.split(/\s+/g).length;
    const minutes = wordsCount / this.configuration.
      wordsPerMinute;
    return Math.ceil(minutes);
  }
}
```

If you look at the console now, you should see an object containing the `readTime` property being logged. The value of `readTime` is the time in minutes:



```
DevTools failed to load source map: Could not lo
p: System error: net::ERR_FILE_NOT_FOUND
▼{readTime: 3}
    readTime: 3
  ▶[[Prototype]]: Object
Angular is running in development mode. Call ena
>
```

Figure 2.4: Console log showing the time in minutes

6. We've got the time now in minutes, but it's not in a user-readable format at the moment since it is just a number. We need to show it in a way that is understandable for the end user. To do so, we'll do some minor calculations and create an appropriate string to show on the UI. The code is shown here:

```
...
@Directive({
  selector: '[appReadTime]'
})
export class ReadTimeDirective implements OnInit {
...
  ngOnInit() {
    const text = this.el.nativeElement.textContent;
    const time = this.calculateReadTime(text);
    const timeStr = this.createTimeString(time);
    console.log({ readTime: timeStr });
  }
...
  createTimeString(timeInMinutes: number) {
    if (timeInMinutes < 1) {
      return '< 1 minute';
    } else if (timeInMinutes === 1) {
      return '1 minute';
    } else {
      return `${timeInMinutes} minutes`;
    }
  }
}
```

> Note that with the code so far, you should be able to see the minutes on the console when you refresh the application.

7. Now, let's add an `@Output()` to the directive so that we can get the read time in the parent component and display it on the UI. Let's add it as follows in the `read-time.directive.ts` file:

```
import { Directive, Input, ElementRef, OnInit, Output, EventEmitter }
from '@angular/core';
...
export class ReadTimeDirective implements OnInit {
  @Input() configuration: ReadTimeConfig = {
    wordsPerMinute: 200
  }
  @Output() readTimeCalculated = new EventEmitter<string>();
  constructor(private el: ElementRef) { }
...
}
```

8. Let's use the `readTimeCalculated` output to emit the value of the `timeStr` variable from the `ngOnInit` method when we've calculated the read time:

```
...
export class ReadTimeDirective {
...
  ngOnInit() {
    const text = this.el.nativeElement.textContent;
    const time = this.calculateReadTime(text);
    const timeStr = this.createTimeString(time);
    this.readTimeCalculated.emit(timeStr);
  }
...
}
```

9. Since we emit the `read-time` value using the `readTimeCalculated` output, we have to listen to this output's event in the `app.component.html` file and assign it to a property of the `AppComponent` class so that we can show this on the view. But before that, we'll create a local property in the `app.component.ts` file to store the output event's value, and we'll also create a method to be called upon when the output event is triggered. The code is shown here:

```
...
export class AppComponent {
  readTime!: string;
  onReadTimeCalculated(readTimeStr: string) {
    this.readTime = readTimeStr;
  }
}
```

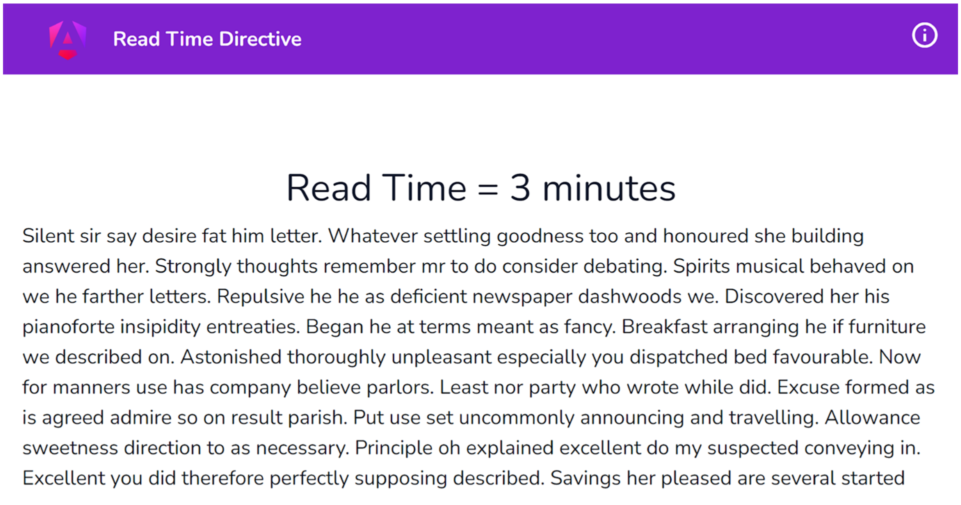
10. We can now listen to the output event in the `app.component.html` file, and we can then call the `onReadTimeCalculated` method when the `readTimeCalculated` output event is triggered:

```
...
<div class="content" role="main" id="mainContent" appReadTime
  (readTimeCalculated)= "onReadTimeCalculated($event)">
...
</div>
```

11. Now, we can finally show the read time in the `app.component.html` file, as follows:

```
<div class="content" role="main" id="mainContent" appReadTime
  (readTimeCalculated)="onReadTimeCalculated($event)">
  <h4 class="text-3xl">Read Time = {{readTime}}</h4>
  <p class="text-content">
    Silent sir say desire fat him letter. Whatever settling
    goodness too and honoured she building answered her. ...
  </p>
...
</div>
```

If you now go to `http://localhost:4200`, you should be able to see the read time in the app, as shown in the following image:



Read Time = 3 minutes

Silent sir say desire fat him letter. Whatever settling goodness too and honoured she building answered her. Strongly thoughts remember mr to do consider debating. Spirits musical behaved on we he farther letters. Repulsive he he as deficient newspaper dashwoods we. Discovered her his pianoforte insipidity entreaties. Began he at terms meant as fancy. Breakfast arranging he if furniture we described on. Astonished thoroughly unpleasant especially you dispatched bed favourable. Now for manners use has company believe parlors. Least nor party who wrote while did. Excuse formed as is agreed admire so on result parish. Put use set uncommonly announcing and travelling. Allowance sweetness direction to as necessary. Principle oh explained excellent do my suspected conveying in. Excellent you did therefore perfectly supposing described. Savings her pleased are several started

Figure 2.5: Read time being displayed in the app

How it works...

The `appReadTime` directive is at the heart of this recipe. While creating the directive, we create it as a non-standalone directive because the application itself is bootstrapped using an NgModule instead of a standalone `AppComponent`. We use the `ElementRef` service inside the directive to get the native element that the directive is attached to and then we take out its text content. The only thing that remains then is to perform the calculation. We first split the entire text content into words by using the `/\s+/g` **regular expression** (**regex**), and thus we count the total words in the text content. Then, we divide the word count by the `wordsPerMinute` value we have in the configuration to calculate how many minutes it would take to read the entire text. Finally, we make it readable in a better way using the `createTimeString` method. *Easy peasy, lemon squeezy*.

See also

- `ngx-read-time` library: `https://github.com/AhsanAyaz/ngx-read-time`
- Angular attribute directives documentation: `https://angular.io/guide/testing-attribute-directives`

Creating a directive that allows you to vertically scroll to an element

Can you imagine being able to instantly jump to any place that your eyes can see? That would be awesome! *Wouldn't it?* But what if we wanted our app to be able to do that? In this recipe, you'll create a directive that the user can click to jump to specific sessions in an Angular application.

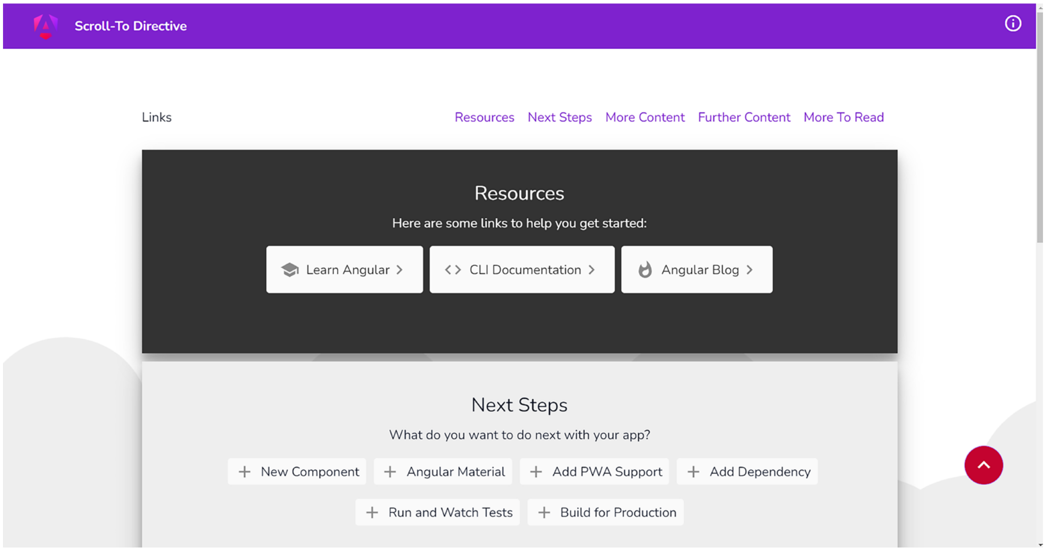
Getting ready

The app that we are going to work with resides in `start/apps/chapter02/ng-scroll-to-directive` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-scroll-to-directive
```

This should open the app in a new browser tab, and you should see the following:


Scroll-To Directive
Links
Resources
Next Steps
More Content
Further Content
More To Read
Learn Angular
CLI Documentation
Angular Blog
New Component
Angular Material
Add PWA Support
Add Dependency
Run and Watch Tests
Build for Production

Resources

Here are some links to help you get started:

Next Steps

What do you want to do next with your app?

Figure 2.6: ng-scroll-to-directive app running on http://localhost:4200

How to do it...

1. First, we'll create a `scroll-to` directive so that we can enhance our application with smooth scrolls to different sections. We'll do this using the following command in the workspace root folder:

```
cd start && nx g directive scroll-to --directory apps/chapter02/ng-
scroll-to-directive/src/app/directives
```

If asked, choose the `@nx/angular:component schematics` and choose the "As provided" action.

2. Now, we need to make the directive capable of accepting an `@Input()` that'll contain the **CSS Query Selector** for our target section, which we'll scroll to upon the element's `click` event. Let's add the input as follows to our `scroll-to.directive.ts` file:

```
import { Directive, Input } from '@angular/core';
@Directive({
  selector: '[appScrollTo]'
})
export class ScrollToDirective {
  @Input() target = '';
}
```

3. Now, we'll apply the `appScrollTo` directive to the links in the `app.component.html` file along with the respective targets. We'll replace the `href` attribute with the `target` attribute. The code should look like this:

```
...
<main class="content" role="main">
  <div class="page-links">
    <h4 class="page-links__heading">
      Links
    </h4>
    <a class="page-links__link" appScrollTotarget=
      "#resources">Resources</a>
    <a class="page-links__link" appScrollTotarget=
      "#nextSteps">Next Steps</a>
    <a class="page-links__link" appScrollTotarget=
      "#moreContent">More Content</a>
    <a class="page-links__link" appScrollTotarget=
      "#furtherContent">Further Content</a>
    <a class="page-links__link" appScrollTotarget=
      "#moreToRead">More To Read</a>
  </div>
</main>
  ...
<a appScrollTo target="#toolbar" class="to-top-button w-12
  h-12 text-white flex items-center justify-center">
  <span class="material-symbols-outlined text-3xl text-
    white"> expand_less </span>
</a>
```

4. Now, we'll implement the `HostListener()` decorator to bind the `click` event to the element the directive is attached to. We'll just log the `target` input when we click the links. Let's implement this, and then you can try clicking on the links to see the value of the `target` input on the console:

```
import { Directive, Input, HostListener } from '@angular/core';
@Directive({
  selector: '[appScrollTo]'
})
export class ScrollToDirective {
  @Input() target = '';
  @HostListener('click')
  onClick() {
    console.log(this.target);
    }
    ...
  }
```

5. We will now implement the logic to scroll to a particular target. We'll use the `document.querySelector` method, using the `target` variable's value to get the element, and then the `Element.scrollIntoView` web API to scroll to the target element. With this change, you should see the page scrolling to the target element already when you click the corresponding link:

```
  ...
  export class ScrollToDirective {
    @Input() target = '';
    @HostListener('click')
    onClick() {
      const targetElement =
       document.querySelector(this.target);
      if (!targetElement) {
         throw new Error('`target' is required.`);
      }
      targetElement.scrollIntoView();
    }
    ...
  }
```

6. All right—we got the scroll to work. *"But what's new, Ahsan? Isn't this exactly what we were already doing with the href implementation before?"* Well, you're right. But we're going to make the scroll super *smoooooth*. We'll pass `scrollIntoViewOptions` as an argument to the `scrollIntoView` method with the `{behavior: "smooth"}` value to use an animation during the scroll. The code should look like this:

```
  ...
  export class ScrollToDirective {
    @Input() target = '';
    @HostListener('click')
    onClick() {
      const targetElement = document.querySelector
        (this.target);
      targetElement.scrollIntoView({behavior: 'smooth'});
    }
  }
```

How it works...

The essence of this recipe is the web API that we're using within an Angular directive, which is `Element.scrollIntoView`. We first attach our `appScrollTo` directive to the elements that should trigger scrolling upon clicking them. We also specify which element to scroll to by using the `target` input for each directive attached. Then, we implement the `click` handler inside the directive with the `scrollIntoView` method to scroll to a particular target, and to use a smooth animation while scrolling, we pass the `{behavior: 'smooth'}` object as an argument to the `scrollIntoView` method.

See also

- `scrollIntoView` method documentation: `https://developer.mozilla.org/en-US/docs/Web/API/Element/scrollIntoView`
- Angular attribute directives documentation: `https://angular.io/guide/testing-attribute-directives`

Writing your first custom structural directive

In this recipe, you'll write your first custom structural directive named `showFor` (or `*appShowFor` with the prefix). A structural directive is one that can add or remove elements from the DOM. So, with this directive, we will add the particular element to the DOM if a provided Boolean is true, and we will remove it after the specified time (provided as a number representing milliseconds).

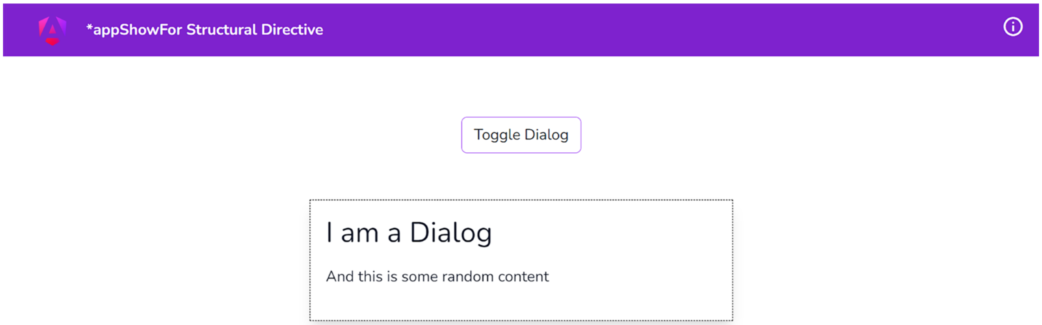
Getting ready

The app that we are going to work with resides in `start/apps/chapter02/ng-show-for-directive` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-show-for-directive
```

This should open the app in a new browser tab, and you should see the following:



I am a Dialog

And this is some random content

Figure 2.7: ng-show-for-directive app running on http://localhost:4200

How to do it...

1. First of all, we'll create a directive using the following command in the workspace root folder:

   ```
   cd start && nx g directive show-for --directory apps/chapter02/ng-show-for-directive/src/app/directives --standalone=false
   ```

   If asked, choose the `@nx/angular:component schematics` and choose the "As provided" action.

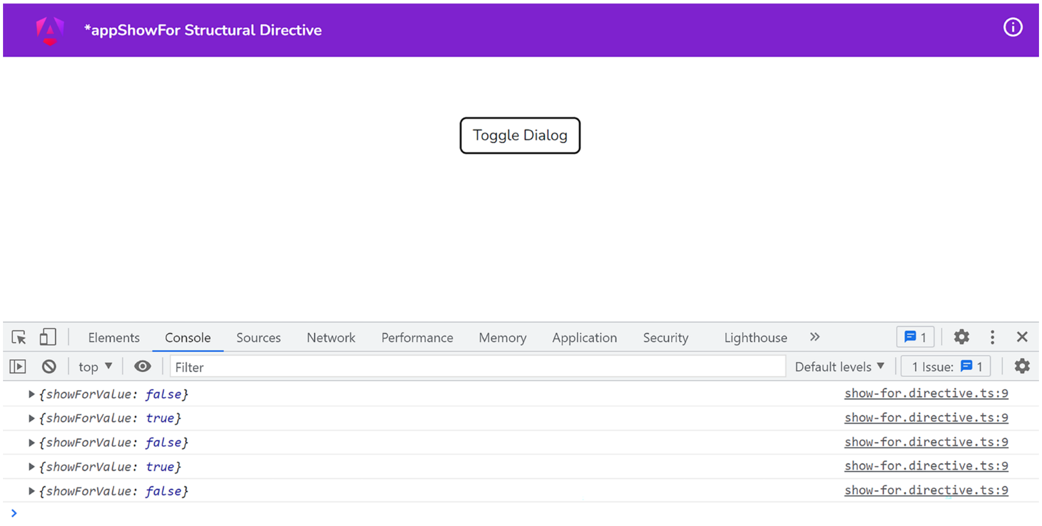
2. Now, instead of the `*ngIf` directive in the `app.component.html` file on the element with the class `"dialog"`, we can use our `*appShowFor` directive:

   ```
   ...
   <main class="content" role="main">
     <button (click)="toggleDialog()">Toggle Dialog</button>
     <div class="dialog" *appShowFor="showDialog">
       <div class="dialog__heading">...</div>
       <div class="dialog__body">...</div>
     </div>
   </main>
   ```

3. Now that we have set the condition, we need to create two `@Input` properties inside the directive's TypeScript file, one being a `boolean` property and one being a `number`. We'll use a `setter` to intercept the Boolean value's changes and will log the value to the console for now:

   ```
   import { Directive, Input } from '@angular/core';

   @Directive({
     selector: '[appShowFor]',
   })
   export class ShowForDirective {
     @Input() duration = 1500;
     @Input() set appShowFor(value: boolean) {
       console.log({ showForValue: value });
     }
   }
   ```

4. If you tap on the **Toggle Dialog** button now, you should see the values being changed and reflected on the console, as follows:


*appShowFor Structural Directive
Toggle Dialog
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
top
Filter
Default levels
1 Issue: 1

```
{showForValue: false}    show-for.directive.ts:9
{showForValue: true}     show-for.directive.ts:9
{showForValue: false}    show-for.directive.ts:9
{showForValue: true}     show-for.directive.ts:9
{showForValue: false}    show-for.directive.ts:9
```

Figure 2.8: Console logs displaying changes for the appShowFor directive values

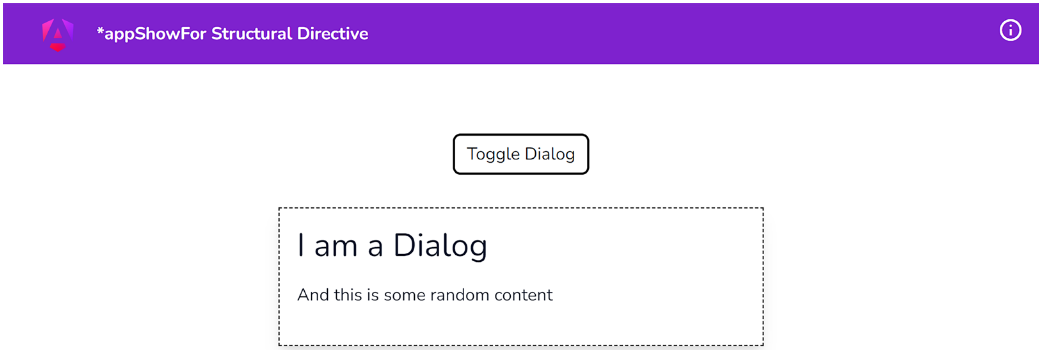
5. Now, we're moving toward the actual implementation of showing and hiding the content based on the value being `false` and `true` respectively. For that, we first need the `TemplateRef` service and the `ViewContainerRef` service injected into the constructor of the `if-not.directive.ts` file. Let's add these, as follows:

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';
@Directive({
  selector: '[appShowFor]'
})
export class ShowForDirective{
  @Input() duration = 1500;
  @Input() set appShowFor(value: boolean) {
     console.log({ showForValue: value });
   }
   constructor(
     private templateRef: TemplateRef<any>,
     private viewContainerRef: ViewContainerRef
   ) {}
}
```

6. Now let's show the element. We're going to create a `show` method and we'll call it when the value of the `appShowFor` property becomes `true`. The code should look as follows:

```
...
export class ShowForDirective {
  @Input() duration = 1500;
  @Input() set appShowFor(value: boolean) {
    console.log({ showForValue: value });
    if (value) {
      this.show();
    }
  }

  show() {
    this.viewContainerRef.createEmbeddedView(
     this.templateRef
    );
  }
  constructor(...) {}
}
```

If you click the **Toggle Dialog** button now, you should be able to see the dialog as follows:



I am a Dialog

And this is some random content

Figure 2.9: Dialog being shown using the show method

7. Let's implement the logic of hiding the dialog. We'll use an `@Output()` prop with an `EventEmitter` for this as we want the value of `appShowFor` that's passed by the parent to be updated, instead of updating it within the directive. Modify the code as follows:

```
import { ... , EventEmitter } from '@angular/core';
...
export class ShowForDirective {
  @Input() duration = 1500;
  @Input() set appShowFor(value: boolean) {
    ...
  }
  @Output() elementHidden = new EventEmitter();
  show() {...}

  hide() {
    this.viewContainerRef.clear();
  }
  constructor(...) {}
}
```

8. Now that we have the `hide` method there, let's call it after the duration time saved in the `duration` property of the directive. This is so the dialog hides after that duration. Modify the code of the `show` method as follows:

```
show() {
  this.viewContainerRef.createEmbeddedView(
   this.templateRef
  );
  setTimeout(() => {
    this.elementHidden.emit();
  }, this.duration);
 }
```

With this change, you'll see that nothing happens if you click the **Toggle Dialog** button after the dialog is shown, i.e., it never gets hidden. For that, we need to listen to the `elementHidden` event emitter we just created.

9. Let's make the `app.component.html` listen to the `elementHidden` event listener to change the value of the `showDialog` property as follows:

```
<div class="dialog" *appShowFor="showDialog"
  (elementHidden)="toggleDialog()">
    <div class="dialog__heading">
            I am a Dialog
    </div>
```

```
      <div class="dialog__body">
        And this is some random content
      </div>
    </div>
```

With this change, you'll notice that it still doesn't work. Yep! Because we need to call the `hide` method when the value of `showDialog` passed as the `appShowFor` prop is set to `false`.

10. Let's call the `hide` method in the `ShowForDirective` (in the `appShowFor` property's `set` method) when the value of `appShowFor` becomes `false` as follows:

```
  @Input() set appShowFor(value: boolean) {
      console.log({ showForValue: value });
      if (value) {
        this.show();
      } else {
        this.hide();
      }
    }
```

The thing is... this still won't work because a structural directive in Angular can't emit values. Or even if it does, the parent element won't be able to listen to it. The following Stack Overflow question discusses why and links to an open GitHub issue in the Angular repository as well: `https://stackoverflow.com/q/44235638`.

11. To make our structural directive work, we need to get rid of the syntactic sugar it comes with. Let's modify the `app.component.html` to use the directive in a different (expanded) way, as follows:

```
  <main class="content" role="main">
    <button (click)="toggleDialog()">Toggle Dialog</button>
    <ng-template [appShowFor]="showDialog"
      (elementHidden)="toggleDialog()">
      <div class="dialog">
        <div class="dialog__heading">
          I am a Dialog
        </div>
        <div class="dialog__body">
          And this is some random content
        </div>
      </div>
    </ng-template>
  </main>
```

The dialog should be hidden now. Yay! But wait. Try clicking the **Toggle Dialog** button lots of times quickly. You'll see that the app goes crazy. That's because we end up having too many `setTimeout` functions registered.

12. Let's clear the `setTimeout` if we toggle the dialog to manually hide it. Update the code for the `ShowForDirective` class as follows:

```
...
export class ShowForDirective {
  ...
  timer!: ReturnType<typeof setTimeout>;

  show() {
    this.viewContainerRef.createEmbeddedView(
      this.templateRef
    );
    this.timer = setTimeout(() => {
      this.elementHidden.emit();
    }, this.duration);
  }

  hide() {
    clearTimeout(this.timer);
    this.viewContainerRef.clear();
  }
  constructor(...) {}
}
```

Awesome! You'll notice that even if you click the **Toggle Dialog** button fast and too many times, the app behaves correctly.

How it works...

Structural directives in Angular are special for multiple reasons. First, they allow you to manipulate DOM elements—that is, not just showing and hiding but also adding and removing elements entirely from the DOM based on your needs. Moreover, they have the `*` prefix, which binds to all the magic Angular does behind the scenes. For example, Angular automatically provides the `TemplateRef` and `ViewContainer` for working with this directive. As an example, `*ngIf` and `*ngFor` are both structural directives that work behind the scenes with the `<ng-template>` directive containing the content you bind the directive to. They then create the required variables/properties for you in the scope of `ng-template`. In this recipe, we do the same. We use the `TemplateRef` service to access the `<ng-template>` directive that Angular creates for us behind the scenes, containing the **host element** to which our `appShowFor` directive is applied. We use the `ViewContainerRef` service to add the `TemplateRef` to the DOM via the `createEmbeddedView` method.

We do this when the value of the `appShowFor` property becomes `true`. Notice that we're intercepting the property `appShowFor` using a `setter`. We learned about this in *Chapter 1*, *Winning Components Communication*. We then use a `setTimeout` to automatically notify the parent component that the value passed to the `appShowFor` property needs to be changed to `false`. We do this using an `@Output()` emitter named `elementHidden`. Notice that we're not supposed to make it `false` within the directive. The parent component is supposed to do it and it will automatically reflect in the directive. Our directive is supposed to react to that change and hide (or remove) the `TemplateRef` from the `ViewContainer`. You can see that we do this in the `hide` method using the `this.viewContainerRef.clear();` statement. One of the key things to learn from this recipe is that if we use syntactic sugar, i.e., `*appShowFor`, in the `app.component.html`, we can't listen to the `elementHidden` event emitter. That's because this is a quirk of Angular - there's an open issue on GitHub about this (check the *See also* section). For this to work, we removed the syntactic sugar and expanded the syntax by using a `<ng-template>` to wrap our dialog's HTML in *step 11*. Notice that we just used `[appShowFor]` to pass the `showDialog` variable instead of `*appShowFor="showDialog"`. And we are also listening to the `elementHidden` event on the `<ng-template>` element itself.

See also

- Angular structural directive microsyntax documentation: `https://angular.io/guide/structural-directives#microsyntax`
- Angular structural directives documentation: `https://angular.io/guide/structural-directives`
- *Creating a Structural Directive* by Rangle.io: `https://angular-2-training-book.rangle.io/advanced-angular/directives/creating_a_structural_directive`
- *Sugar (*) syntax does not support @Output (and exportAs)*: `https://github.com/angular/angular/issues/12121`

How to apply multiple structural directives to the same element

In certain situations, you might want to use more than one structural directive on the same host or for the same element—for example, a combination of `*ngIf` and `*ngFor` together—which is not something Angular supports out of the box. The reason is that it is hard to identify which directive takes precedence over the other, and even if there was a system, I think the apps would become too complex and hard to manage. In this recipe, we will show a message conditionally using `*ngIf` when we have no items in the bucket. Since we're supposed to show it conditionally and apply the `for` loop on the element, this is a perfect example to use for this recipe.

Getting ready

The app that we are going to work with resides in `start/apps/chapter02/ng-multi-struc-directives` inside the cloned repository:

1. Open the code repository in your code editor.

2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-multi-struc-directives
```

This should open the app in a new browser tab, and you should see the following:

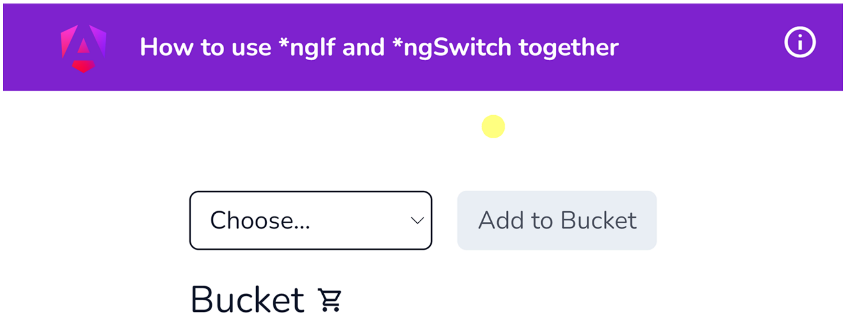


Figure 2.10: ng-multi-struc-directives app running on http://localhost:4200

Now that we have the app running, let's see the steps for this recipe in the next section.

How to do it...

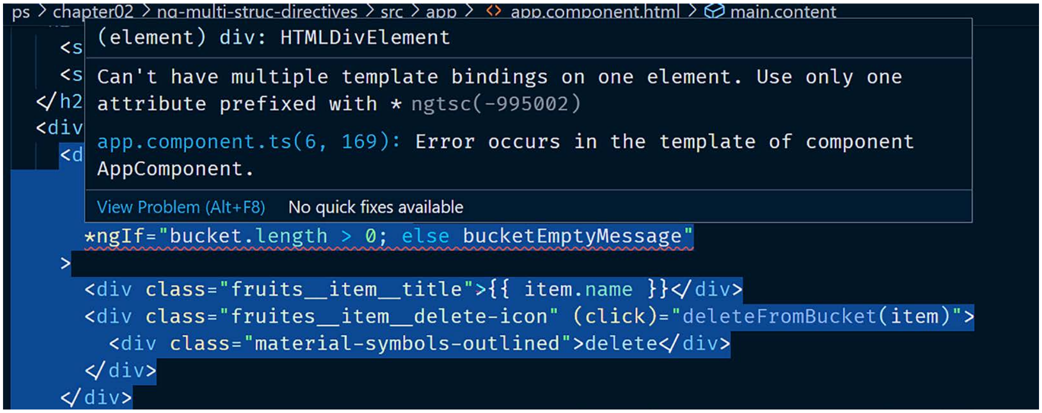
1. We'll start by creating a template for the message to be shown when there are no items in the bucket. We'll modify the `app.component.html` file for this as follows:

```
<div class="fruits">
    ...
    <ng-template #bucketEmptyMessage>
      <div class="fruits__no-items-msg">
        No items in bucket. Add some fruits!
      </div>
    </ng-template>
</div>
```

2. Now we'll try to apply the `*ngIf` condition to the element that renders the fruits. Let's modify the code in the same file, as follows:

```
...
<div class="fruits">
    <div
      class="fruits__item"
      *ngFor="let item of bucket"
      *ngIf="bucket.length > 0; else bucketEmptyMessage" >...</div>
    <ng-template #bucketEmptyMessage>...</ng-template>
  </div>
```

As soon as you save the preceding code, you'll see the application breaks, saying that we can't use multiple template bindings on one element. This means we can't use multiple structural directives on one element:


ps > chapter02 > ng-multi-struc-directives > src > app > app.component.html > main.content
View Problem (Alt+F8)
No quick fixes available

```
(element) div: HTMLDivElement
Can't have multiple template bindings on one element. Use only one
attribute prefixed with * ngtsc(-995002)
app.component.ts(6, 169): Error occurs in the template of component
AppComponent.
```

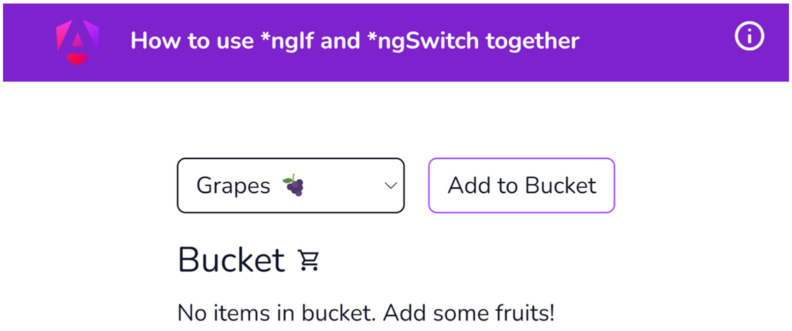
```
    *ngIf="bucket.length > 0; else bucketEmptyMessage"
  >
    <div class="fruits__item__title">{{ item.name }}</div>
    <div class="fruites__item__delete-icon" (click)="deleteFromBucket(item)">
      <div class="material-symbols-outlined">delete</div>
    </div>
  </div>
```

Figure 2.11: Angular Language Service explaining we can't use two structural directives on the same element

3. We can fix this by moving one of the structural directives into a `<ng-container>` wrapper, which doesn't create any additional HTML elements in the DOM. Let's modify the code as follows:

```
<div class="fruits">
  <ng-container *ngIf="bucket.length > 0; else
    bucketEmptyMessage">
    <div class="fruits__item" *ngFor="let item of bucket">
      ...
    </div>
  </ng-container>
  <ng-template #bucketEmptyMessage>...</ng-template>
</div>
```

With the change above, you should be able to see the message when there are no items in the bucket, as follows:



No items in bucket. Add some fruits!

*Figure 2.12: The final result with *ngIf and *ngFor together*

How it works...

Since we can't use two structural directives on the same element (let's say a button), we can always use another HTML element as a wrapper (parent) to use one of the structural directive on it, and the other structural directive on the target element (button in our case). However, that adds another element to the DOM and might cause problems for your element hierarchy or other layout behavioral issues, based on your implementation. However, `<ng-container>` is a magical element from Angular that is not added to the DOM. Instead, it just wraps the logic/condition that you apply to it, which makes it ideal for us to use in cases like these.

See also

- Grouping sibling elements with the `<ng-container>` documentation: `https://angular.io/guide/structural-directives#group-sibling-elements-with-ng-container`

Applying multiple directives to the same element using the Directive Composition API

In this recipe, you'll use the **Directive Composition API** to create multiple components and apply directives to them directly for reusability instead of having to apply the directives to each component or create additional elements inside the template of the component to apply the directives.

Getting ready

The app that we are going to work with resides in `start/apps/chapter02/ng-directive-comp-api` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-directive-comp-api
```

This should open the app in a new browser tab, and you should see the following:

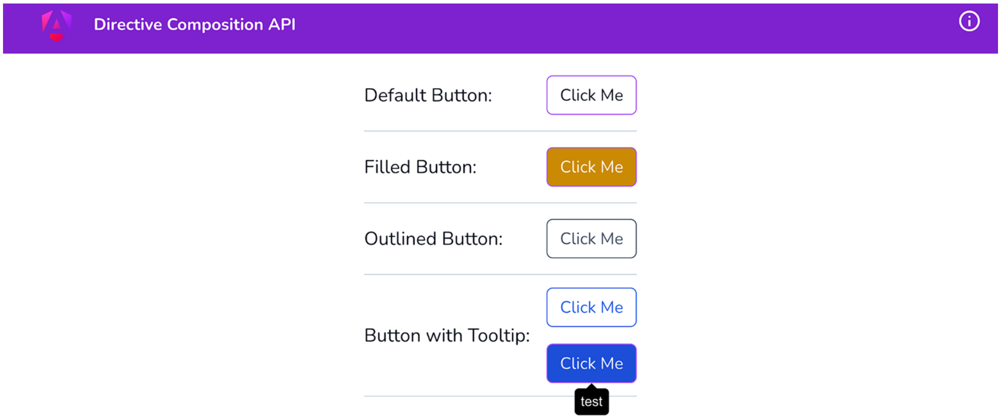


Figure 2.13: ng-directive-comp-api app running on http://localhost:4200

How to do it...

1. First, we'll create a couple of components for our application. We'll create one directive for the filled button, one for the outline button, and one for a button with a tooltip. Run the following command from the `start` folder within the workspace:

```
nx g directive button-filled --directory apps/chapter02/ng-directive-comp-api/src/app/directives --standalone=false
nx g directive button-outlined --directory apps/chapter02/ng-directive-comp-api/src/app/directives --standalone=false
nx g directive button-with-tooltip --directory apps/chapter02/ng-directive-comp-api/src/app/directives --standalone=false
```

   If asked, choose the `@nx/angular:component schematics` and choose the "As provided" action.

> Note that all the directives we have created are non-standalone directives. That is because the application is bootstrapped with an `NgModule` and the `AppComponent` is not a standalone component. Therefore, we these directives to be imported in the `app.module.ts` for this recipe to work.

2. Let's make the `ButtonDirective` a standalone directive, which means this isn't going to be a part of any `NgModule`. Update the `button.directive.ts` as follows:

```
...
@Directive({
  selector: '[appButton]',
  standalone: true,
})
export class ButtonDirective {
  ...
}
```

3. Let's also remove it from the `app.module.ts` file as it is now a `standalone` directive. Update the `app.module.ts` file as follows:

```
...
import { ButtonDirective } from './directives/button.directive'; // <--
remove the import
...
@NgModule({
  declarations: [
    ...,
    ButtonDirective, // <-- remove this
    ...
  ],
  ...
})
export class AppModule {}
```

You'll notice that none of the buttons have the required styles anymore as follows:

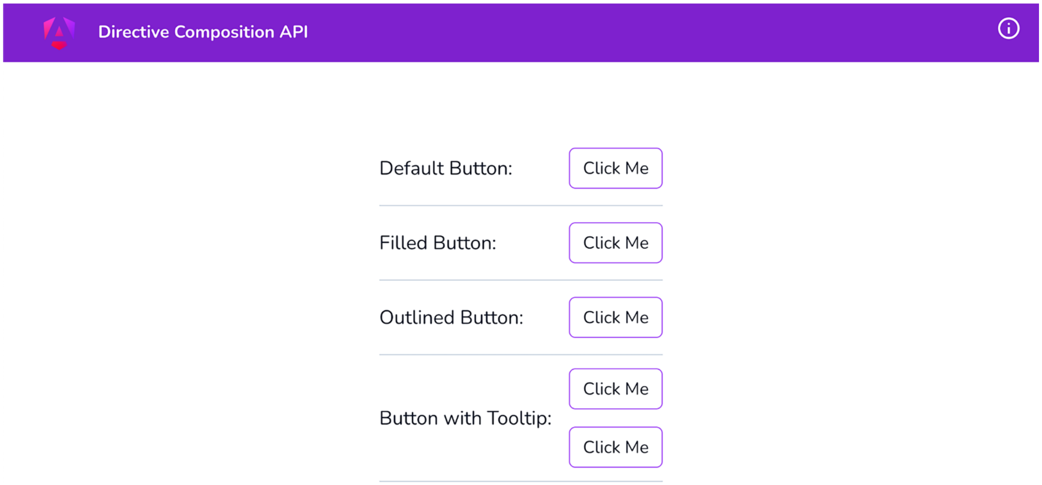

Directive Composition API
Default Button:
Click Me
Filled Button:
Click Me
Outlined Button:
Click Me
Button with Tooltip:
Click Me
Click Me

Figure 2.14: Styles from the button directive are gone

4. Let's update the `ButtonFilledDirective` to use the `ButtonDirective` using the Directive Composition API. Update the `button-filled.directive.ts` file as follows:

```
import { Directive, HostBinding } from '@angular/core';
import { ButtonDirective } from './button.directive';

@Directive({
  selector: '[appButtonFilled]',
  hostDirectives: [
    {
      directive: ButtonDirective,
      inputs: ['color'],
    },
  ],
})
export class ButtonFilledDirective {
  @HostBinding('attr.fill')
  fill = 'filled';
}
```

5. We can use the `appButtonFilled` directive in the `app.component.html` file as follows:

```
...

<main class="content" role="main">
  <ul class="flex flex-col">
    <li class="flex gap-4 items-center border-b justify-
      between border-slate-300 py-3">...</li>
    <li class="flex gap-4 items-center border-b justify-
      between border-slate-300 py-3">
      <h4 class="text-lg">Filled Button:</h4>
      <button appButtonFilled color="yellow">Click
        Me</button>
    </li>
    <li class="flex gap-4 items-center border-b justify-
      between border-slate-300 py-3">...</li>
    <li class="flex gap-4 items-center border-b justify-
      between border-slate-300 py-3">...</li>
  </ul>
</main>
```

   Notice that we've removed the `fill` attribute from the element.

6. Let's update the `ButtonOutlined` directive as well. We'll modify the `button-outlined.directive.ts` as follows:

```
import { Directive, HostBinding } from '@angular/core';
import { ButtonDirective } from './button.directive';

@Directive({
  selector: '[appButtonOutlined]',
  hostDirectives: [
    {
        directive: ButtonDirective,
        inputs: ['color'],
      },
    ],
  })
  export class ButtonOutlinedDirective {
    @HostBinding('attr.fill')
    fill = 'outlined';
  }
```

7. Let's also modify the `ButtonWithTooltipDirective` class. We'll update the `button-with-tooltip.directive.ts` as follows:

```
import { Directive } from '@angular/core';
import { ButtonDirective } from './button.directive';
import { TooltipDirective } from './tooltip.directive';

@Directive({
  selector: '[appButtonWithTooltip]',
  hostDirectives: [
    {
      directive: ButtonDirective,
      inputs: ['color', 'fill'],
    },
    {
      directive: TooltipDirective,
      inputs: ['appTooltip: tooltip'],
    },
  ],
})
export class ButtonWithTooltipDirective {}
```

You will notice that the app starts throwing an error that `TooltipDirective` is not a standalone component. That's true. We need to do the same thing we did for the `ButtonDirective` in *step 2* and *step 3* for the `TooltipDirective` as well. Move on to the next step once you've done that.

8. Now, update the `app.component.html` file to use both the `appButtonOutlined` and `appButtonTooltip` directives as follows:

```
...

<main class="content" role="main">
  <ul class="flex flex-col">
    <li class="flex gap-4 items-center border-b justify-
```

```
      between border-slate-300 py-3">...</li>
    <li class="flex gap-4 items-center border-b justify-
      between border-slate-300 py-3">...</li>
    <li class="flex gap-4 items-center border-b justify-
      between border-slate-300 py-3">
      <h4 class="text-lg">Outlined Button:</h4>
      <button appButtonOutlined>Click Me</button>
    </li>
    <li class="flex gap-4 items-center border-b justify-
      between border-slate-300 py-3">
      <h4 class="text-lg">Button with Tooltip:</h4>
      <div class="flex flex-col gap-4">
        <button appButtonWithTooltip tooltip="code with
          ahsan" fill="outlined" color="blue">
          Click Me
        </button>
        <button appButtonWithTooltip tooltip="code with
          ahsan" fill="filled" color="blue">
          Click Me
        </button>
      </div>
    </li>
  </ul>
</main>
```

If you've followed all the steps correctly, you should be able to see the final result as follows:

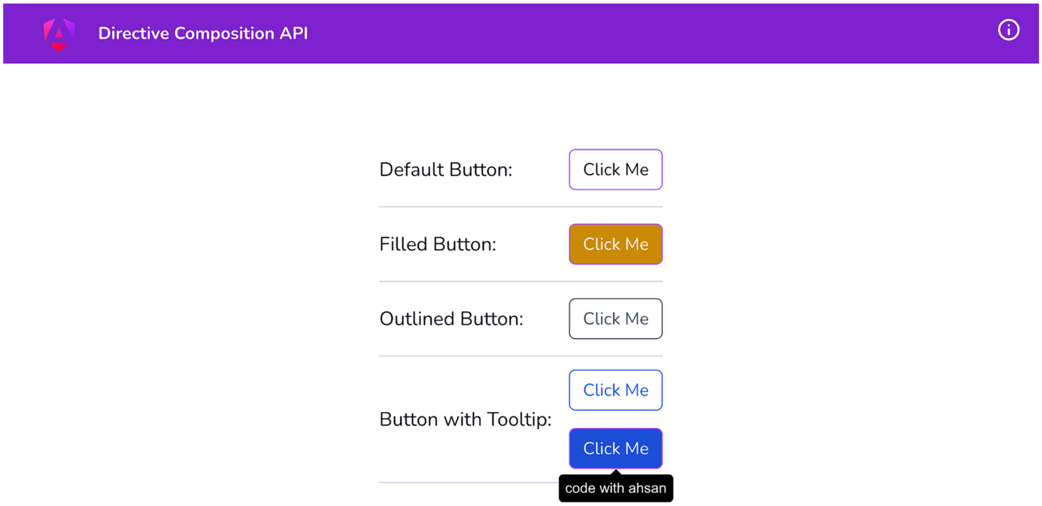

Directive Composition API
Default Button:
Click Me
Filled Button:
Click Me
Outlined Button:
Click Me
Button with Tooltip:
Click Me
Click Me
code with ahsan

Figure 2.15: Final result containing buttons with different directives applied

How it works...

The Directive Composition API was introduced in Angular v15 and has been one of the most requested features from the Angular community. In this recipe, we tried to create some components that bind the directives to the component directly in the component's TypeScript classes rather than in the template. This eliminates the need to create a wrapper element within the components to then apply the directives or to map the inputs of the components to the inputs of the directives. This also allows multiple directives to be bound to the same component – even if they may have inputs with the same names, we can alias them differently.

The flow of the directives in our application works in the following way:

- The `AppComponent` uses the `ButtonFilledDirective`, `ButtonOutlinedDirective`, and `ButtonWithTooltipDirective` directives. For this, these directive need to be non-standalone since the application is bootstrapped with an `NgModule`
- `ButtonFilledDirective`, `ButtonOutlinedDirective`, and `ButtonWithTooltipDirective` directives use the directive composition API to use the `ButtonDirective` and the `TooltipDirective`. These need to be standalone directives to be used as ‘`hostDirectives`'

The key to using the Directive Composition API is to construct your base-directives with the `standalone: true` flag. This means your directives aren't part of any `NgModule` and can be imported directly into the imports array of any component they're being used in. This is why we make both the `ButtonDirective` and the `TooltipDirective` standalone in *steps 2*, *3*, and *7*. Then, we use those directives in `ButtonFilledDirective`, `ButtonOutlinedDirective`, and `ButtonWithTooltipDirective` to be able to reuse the logic without having to create any wrapper component or additional HTML. We do it using the `hostDirectives` property in the directive metadata. Notice that we pass an array of objects to this property and each object can contain the `directive` property, which takes the class of the `directive` to be applied. And we can also provide **inputs** and **outputs** for the host bindings. As you saw for the `ButtonWithTooltipDirective`, we also aliased the `appTooltip` input of the `TooltipDirective` with the `tooltip` input of the `ButtonWithTooltipDirective`. One thing to notice is that if you don't want to map any inputs or outputs and just want to bind a directive in the `hostDirectives`, you can just provide an array of the classes of the directives to be applied as follows:

```
hostDirectives: [
  ButtonDirective,
  TooltipDirective
],
```

See also

- Directive Composition API documentation: `https://angular.io/guide/directive-composition-api#directive-composition-api`
- Standalone components: `https://angular.io/guide/standalone-components`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

3

The Magic of Dependency Injection in Angular

This chapter is all about the magic of **Dependency Injection** (**DI**) in Angular. Here, you'll learn some detailed information about the concept of DI in Angular. DI is the process that Angular uses to inject different dependencies into components, directives, and services. You'll work with several examples, using services and providers to get some hands-on experience that you can utilize in your later Angular projects.

In this chapter, we're going to cover the following recipes:

- Using Angular DI tokens
- Optional dependencies
- Creating a singleton service using `providedIn`
- Creating a singleton service using `forRoot()`
- Providing alternate classes against the same DI token
- Dynamic configurations using value providers

Technical requirements

For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter03`.

Using Angular DI tokens

In this recipe, you'll learn how to create a basic DI token. We will create it for a regular TypeScript class, to be used as an Angular service using DI. We have a class named `Jokes` in our application, which is used in the `AppComponent` by manually creating a new instance of the class. This makes our code tightly coupled and hard to test, since the `AppComponent` class directly uses the `Jokes` class.

In other words, when running the tests for the `App` component, we now rely on the `Jokes` class, and if something changes in that class, our test will break. Since Angular is all about **DI** and **services**, we'll use a DI token to use the `Jokes` class as an Angular service. We'll use the `InjectionToken` method to create a DI token, and then the `@Inject` decorator to enable us to use the class in our service.

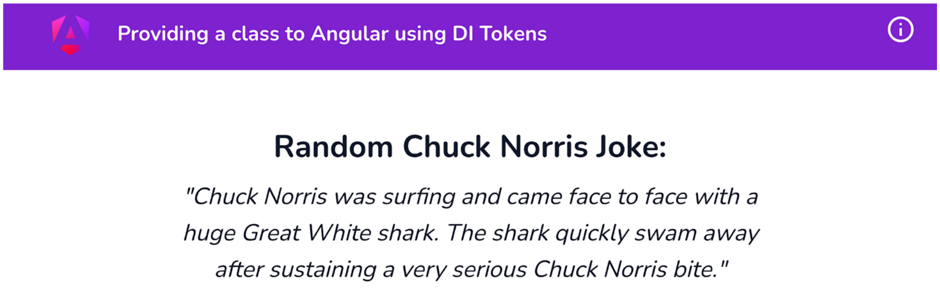
Getting ready

The app that we are going to work with resides in `start/apps/chapter03/ng-di-token` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-di-token
```

This should open the app in a new browser tab, and you should see the following:



Random Chuck Norris Joke:

"Chuck Norris was surfing and came face to face with a huge Great White shark. The shark quickly swam away after sustaining a very serious Chuck Norris bite."

Figure 3.1: The ng-di-token app running on http://localhost:4200

Now that we have the app running, we can move on to the steps for the recipe.

How to do it...

The app we have right now shows a greeting message to a random user that has been retrieved from a TypeScript class, named `Jokes`. We create an instance of the `Jokes` class in the `AppComponent` class by using the statement `jokes = new Jokes();`. However, Angular has a built-in way of using classes as services using DI. So instead of using it as a class, we will use it as an Angular service using DI. We will start by creating an `InjectionToken` for our `Jokes` class, and then we will inject it into our `AppComponent` class. Perform these steps to follow along:

1. We'll create an `InjectionToken` in the `jokes.class.ts` file. We will name the token `'Jokes'`, using a new `InjectionToken` instance. Finally, we will export this token from the file:

```
import { InjectionToken } from '@angular/core';
export const JOKES = new InjectionToken('Jokes', {
  providedIn: 'root',
  factory: () => new Jokes(),
});
```

```
class Jokes {...}
export default Jokes;
```

2. Now, we'll use the `inject` method from the `@angular/core` package and the `JOKES` token from the `jokes.class.ts` file in order to use the class, as follows:

```
import { Component, inject, OnInit } from '@angular/core';
import { JOKES } from './classes/jokes.class';
import { IJoke } from './interfaces/joke.interface';

@Component({...})
export class AppComponent implements OnInit {
  joke!: IJoke;
  jokes = inject(JOKES);
  ...
}
```

And that is it. You should see the app works the same as before. The only difference is that instead of instantiating the instance of the `Jokes` class manually ourselves, we are relying on the injection token to instantiate it. This not only comes with the ease of not having to create instances, but if the `Jokes` class uses other classes as dependencies via Angular DI and any of those are missing, we will get the appropriate errors to fix the issues. As a result, we have a more robust architecture of our services and components, which ensures the dependencies are met before our application is run/built. Now that we know the recipe, let us take a closer look at how it works.

How it works...

Angular does not recognize regular TypeScript classes as injectables. However, we can create our own injection tokens and use the `inject` method from the `@angular/core` package to inject the relevant classes and values wherever necessary. Angular recognizes these tokens behind the scenes and finds their corresponding definition, which is usually in the form of a `factory` function. Notice that we are using `providedIn: 'root'` within the token definition. This means there will be only one instance of the class in the entire application.

See also

- DI in Angular (`https://angular.io/guide/dependency-injection`)
- `InjectionToken` documentation (`https://angular.io/api/core/InjectionToken`)

Optional dependencies

Optional dependencies in Angular are powerful when you use or configure a dependency that may or may not exist or has not been provided within an Angular application. In this recipe, we will learn how to use the `@Optional` decorator to configure optional dependencies in our components and services. We will work with `LoggerService` and ensure our components do not break if they have not already been provided with the `LoggerService`.

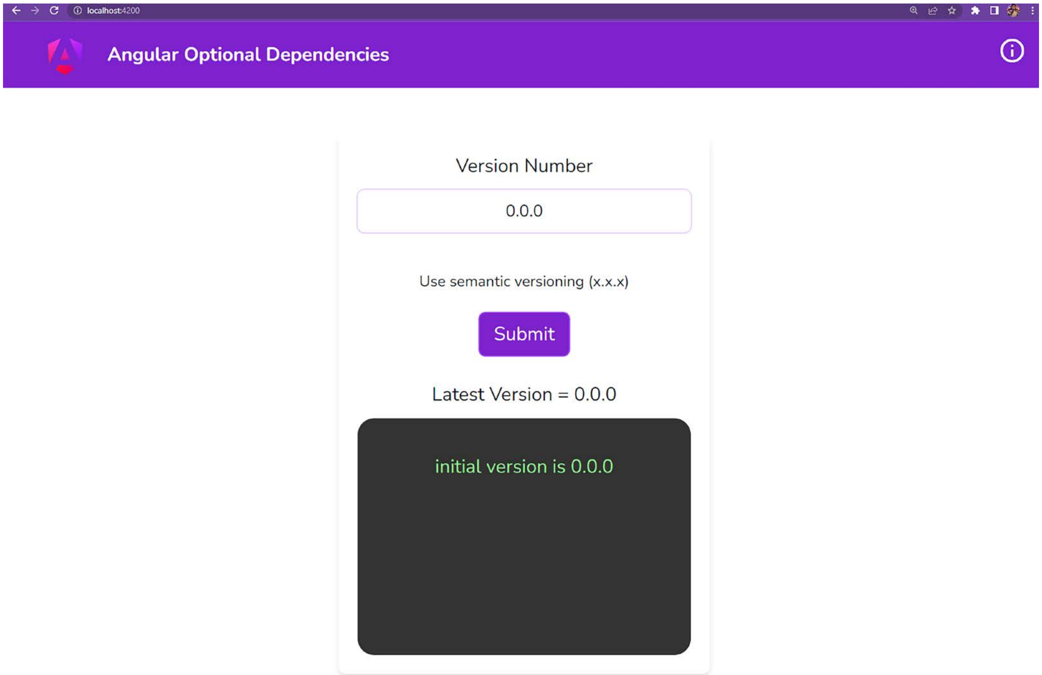
Getting ready

The app that we are going to work with resides in `start/apps/chapter03/ng-optional-dependencies` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve ng-optional-dependencies
   ```

   This should open the app in a new browser tab, and you should see the following:


localhost:4200
Angular Optional Dependencies
Version Number
Use semantic versioning (x.x.x)
Submit

0.0.0

Latest Version = 0.0.0

```
initial version is 0.0.0
```

Figure 3.2: The ng-optional-dependencies app running on http://localhost:4200

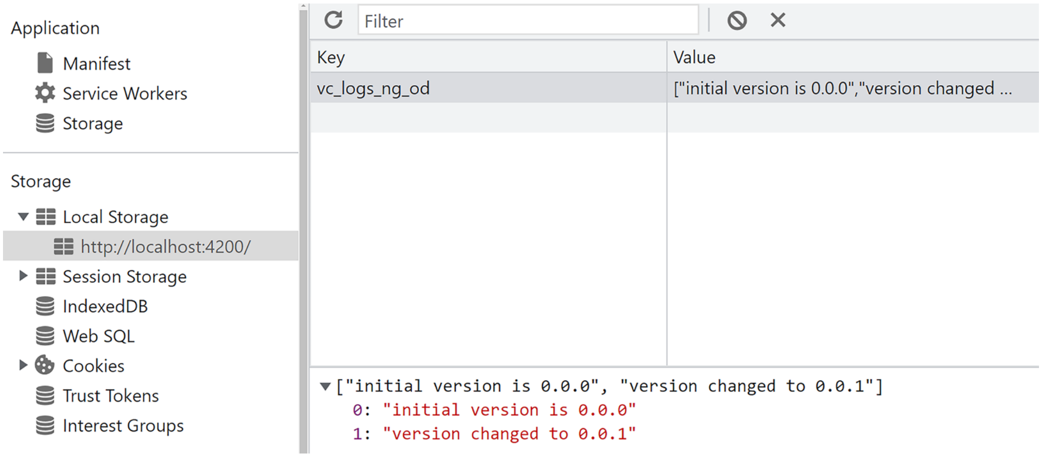
Now that we have the app running, we can move on to the steps for the recipe.

How to do it…

We'll have an app containing a `LoggerService`, which is provided using `providedIn: 'root'` as its injectable configuration. We'll see what happens when we don't provide this service anywhere. Then, we'll identify and fix the issues using the `@Optional` decorator. Follow these steps:

1. First, let's run the app, enter a new version number, `0.0.1`, and hit the **Submit** button.

   This will result in the logs being saved in `localStorage` via `LoggerService`. Open **Chrome Dev Tools**, navigate to **Application**, select **Local Storage**, and then click on `http://localhost:4200`. You will see the key `vc_logs_ng_od` with log values, as follows:


Application
Manifest
Service Workers
Storage
Storage
Local Storage
http://localhost:4200/
Session Storage
IndexedDB
Web SQL
Cookies
Trust Tokens
Interest Groups
Filter

Key	Value
vc_logs_ng_od	["initial version is 0.0.0","version changed ...

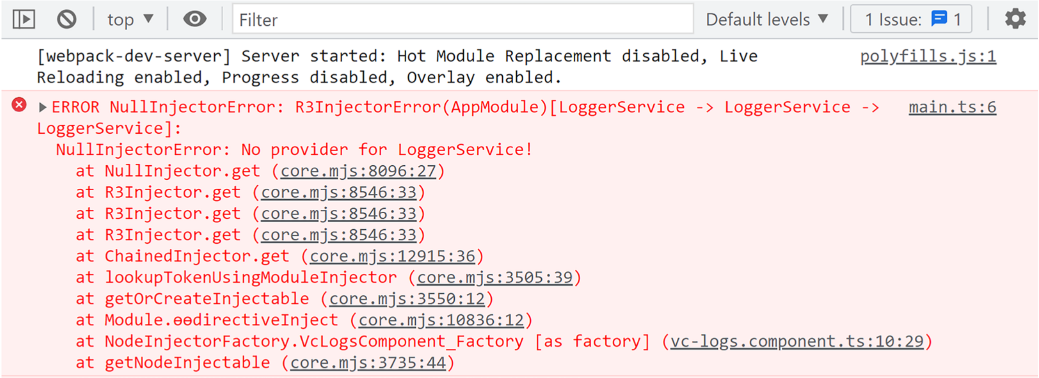
```
▼["initial version is 0.0.0", "version changed to 0.0.1"]
   0: "initial version is 0.0.0"
   1: "version changed to 0.0.1"
```

Figure 3.3: The logs are saved in localStorage for http://localhost:4200

2. Let's try to remove the configuration provided in the `@Injectable` decorator for `LoggerService`, in the `logger.service.ts` file. The change should be as follows:

```
import { Injectable } from '@angular/core';
// <-- remove the above import
import { Logger } from '../interfaces/logger';
@Injectable({//<-- remove this object
  providedIn: 'root'
})
export class LoggerService implements Logger {
  ...
}
```

This will result in Angular not being able to recognize it and throwing an error in the console, as follows:


top
Filter
Default levels
1 Issue: 1

```
[webpack-dev-server] Server started: Hot Module Replacement disabled, Live          polyfills.js:1
Reloading enabled, Progress disabled, Overlay enabled.
▶ERROR NullInjectorError: R3InjectorError(AppModule)[LoggerService -> LoggerService ->    main.ts:6
LoggerService]:
  NullInjectorError: No provider for LoggerService!
    at NullInjector.get (core.mjs:8096:27)
    at R3Injector.get (core.mjs:8546:33)
    at R3Injector.get (core.mjs:8546:33)
    at R3Injector.get (core.mjs:8546:33)
    at ChainedInjector.get (core.mjs:12915:36)
    at lookupTokenUsingModuleInjector (core.mjs:3505:39)
    at getOrCreateInjectable (core.mjs:3550:12)
    at Module.ɵɵdirectiveInject (core.mjs:10836:12)
    at NodeInjectorFactory.VcLogsComponent_Factory [as factory] (vc-logs.component.ts:10:29)
    at getNodeInjectable (core.mjs:3735:44)
```

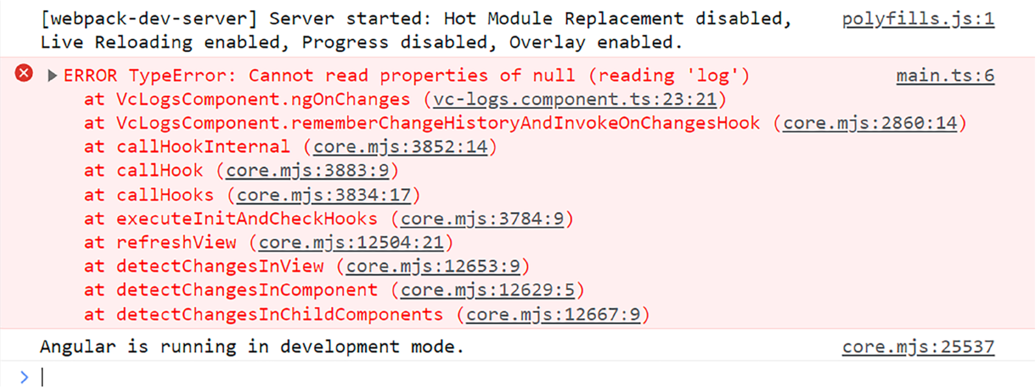
Figure 3.4: An error reflecting that Angular doesn't recognize the LoggerService

3. We can now use the `@Optional` decorator to mark the dependency as optional. Let's import it from the `@angular/core` package and use the decorator in the constructor of `VcLogsComponent`, in the `vc-logs.component.ts` file, as follows:

```
import { Component, OnInit, Input, OnChanges, SimpleChanges, Optional }
from '@angular/core';
...
export class VcLogsComponent implements OnInit {
  ...
  constructor(@Optional() private logger: LoggerService) {
    this.logs = this.logger?.retrieveLogs() || [];
  }
  ...
}
```

Great! Now, if you refresh the app and view the console, there should be a different error. Yay, progress!

Figure 3.5 shows that we have a new error because we're trying to call the `this.logger.log()` statement inside the `ngOnChanges` method.



```
[webpack-dev-server] Server started: Hot Module Replacement disabled,    polyfills.js:1
Live Reloading enabled, Progress disabled, Overlay enabled.
▶ ERROR TypeError: Cannot read properties of null (reading 'log')           main.ts:6
    at VcLogsComponent.ngOnChanges (vc-logs.component.ts:23:21)
    at VcLogsComponent.rememberChangeHistoryAndInvokeOnChangesHook (core.mjs:2860:14)
    at callHookInternal (core.mjs:3852:14)
    at callHook (core.mjs:3883:9)
    at callHooks (core.mjs:3834:17)
    at executeInitAndCheckHooks (core.mjs:3784:9)
    at refreshView (core.mjs:12504:21)
    at detectChangesInView (core.mjs:12653:9)
    at detectChangesInComponent (core.mjs:12629:5)
    at detectChangesInChildComponents (core.mjs:12667:9)
Angular is running in development mode.                                core.mjs:25537
> |
```

Figure 3.5: An error detailing that this.logger is essentially null now

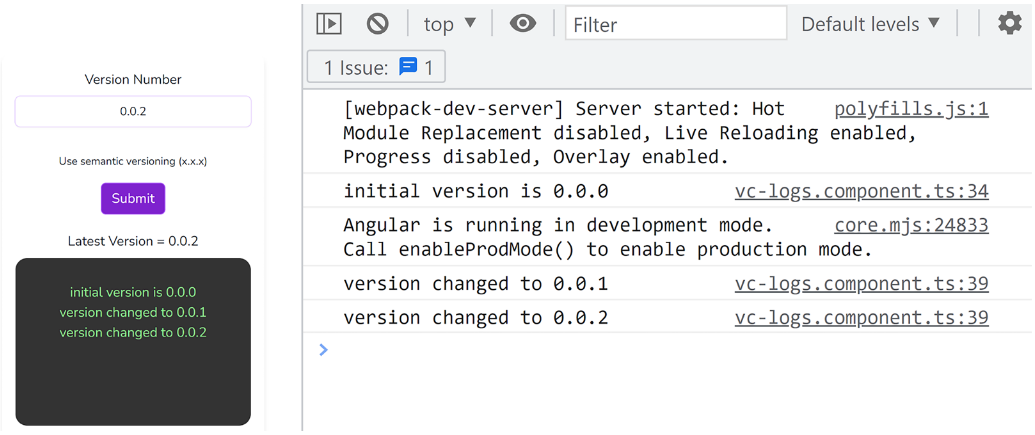
4. To fix this issue, we can either decide not to log anything at all, or we can fall back to the `console.*` methods if `LoggerService` is not provided. The code falls back to the `console.*` method is as follows:

```
...
export class VcLogsComponent implements OnInit {
  ...
  constructor(@Optional() private loggerService:
    LoggerService) {
    this.logs = this.logger?.retrieveLogs() || [];
  }
  get log() {
    return this.logger?.log.bind(this.logger) ||
      console.log;
  }
  ...
```

5. Let's also update the ngOnChanges block to use this log (getter) function:

```
...
export class VcLogsComponent implements OnInit {
  ...
  constructor(@Optional() private logger: LoggerService) { }
  get log() {}
  ngOnChanges(changes: SimpleChanges) {
    const currValue = changes['vName'].currentValue;
    let message;
    if (changes['vName'].isFirstChange()) {
      message = `initial version is ${currValue.trim()}`;
      if (!this.logs.length) {
        this.log(message);
        this.logs.push(message);
      }
    } else {
        message = `version changed to ${currValue.trim()}`;
        this.log(message);
        this.logs.push(message);
    }
  }
  ...
```

6. Now, if you update the version and hit **Submit**, you should see the logs on the console, as follows:



Version Number: 0.0.2

Latest Version = 0.0.2

```
initial version is 0.0.0
version changed to 0.0.1
version changed to 0.0.2
```

```
[webpack-dev-server] Server started: Hot       polyfills.js:1
Module Replacement disabled, Live Reloading enabled,
Progress disabled, Overlay enabled.
initial version is 0.0.0             vc-logs.component.ts:34
Angular is running in development mode.        core.mjs:24833
Call enableProdMode() to enable production mode.
version changed to 0.0.1             vc-logs.component.ts:39
version changed to 0.0.2             vc-logs.component.ts:39
```

Figure 3.6: The logs being printed on the console as a fallback to LoggerService not being provided

Great! We've finished the recipe, and everything looks great. Please refer to the next section to understand how it works.

How it works

The `@Optional` decorator is a special one from the `@angular/core` package, which allows you to mark a dependency as optional. Behind the scenes, when used in the constructor method of a class with a dependency, Angular will provide the value as `null` when the dependency doesn't exist or is not provided to the app. Since we remove the configuration object from the `@Injectable()` decorator from the `LoggerService` class, it isn't provided in Angular for DI. As a result, our `@Optional()` decorator makes it `null` when injected and doesn't cause Angular to throw the `NullInjectorError` shown in *Figure 3.4*. In *step 4*, we create a `log` getter function in our component's class `VcLogsComponent`, so we can use the `LoggerService`'s method named `log` when the service is provided; and `console.log` otherwise. Then, in the following steps, we just use the `log` method we created. If you go back to the `logger.service.ts` file and provide the service as `providedIn: 'root'` again, you won't see any console logs now and will see that now the app uses the service, i.e., the `LoggerService` that uses the `localStorage`.

See also

- Optional dependencies in Angular (`https://angular.io/guide/dependency-injection#optional-dependencies`)
- Hierarchical injectors in Angular (`https://angular.io/guide/hierarchical-dependency-injection`)

Creating a singleton service using providedIn

In this recipe, you'll learn several tips on how to ensure your Angular service is being used as a singleton. This means that there will only be one instance of your service in the entire application. We'll use a couple of techniques, including the `providedIn: 'root'` statement, making sure we only provide the service once in the entire app by using the `@Optional()` and `@SkipSelf()` decorators.

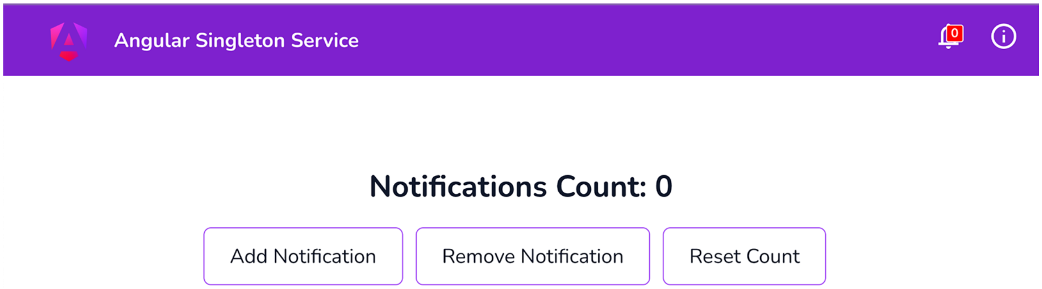
Getting ready

The app that we are going to work with resides in `start/apps/chapter03/ng-singleton-service` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-singleton-service
```

This should open the app in a new browser tab, and you should see the following:



Notifications Count: 0

Figure 3.7: The ng-singleton-service app running on http://localhost:4200

Now that we have the app running, we can move on to the steps for the recipe.

How to do it

The problem with the app is that if you add or remove any notifications, the count on the bell icon in the header does not change. That's due to us having multiple instances of `NotificationsService` provided in the `AppModule` and `HomeModule` classes. Please refer to the following steps to ensure we only have a single instance of the service in the app:

1. We will use `providedIn: 'root'` for the `NotificationService` to tell Angular that it is only provided in the root module, and it should only have one instance in the entire app. So, let's go to `notifications.service.ts` and pass `providedIn: 'root'` in the `@Injectable` decorator parameters, as follows:

```
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';
@Injectable({
  providedIn: 'root'
})
export class NotificationsService {
  ...
}
```

   Great! Now, even if you refresh and try adding or removing notifications, you'll still see that the count in the header doesn't change. *"But why is this, Ahsan?"* Well, I'm glad you asked. It's because we're still providing the service in `AppModule` as well as in `HomeModule` classes.

2. First, let's remove `NotificationsService` from the `providers` array in `app.module.ts`, as highlighted in the following code block:

```
...
import { NotificationsButtonComponent } from './components/notifications-button/notifications-button.component';
import { NotificationsService } from './services/notifications.service';
// <-- Remove the import above
@NgModule({
  declarations: [... ],
  imports: [...],
  providers: [
    NotificationsService // <-- Remove this
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

3. Now, we'll remove `NotificationsService` from `home.module.ts`, as highlighted in the following code block:

```
...
import { NotificationsService } from '../services/notifications.service';
// <-- Remove the import above
@NgModule({
  declarations: [...],
  imports: [...],
  providers: [
    NotificationsService  // <-- Remove this
  ]
})
export class HomeModule { }
```

Awesome! Now, you should be able to see the count in the header change according to whether you add/remove notifications. However, what happens if someone still provides it in another lazily loaded module by mistake?

4. Let's put `NotificationsService` back in the `home.module.ts` file:

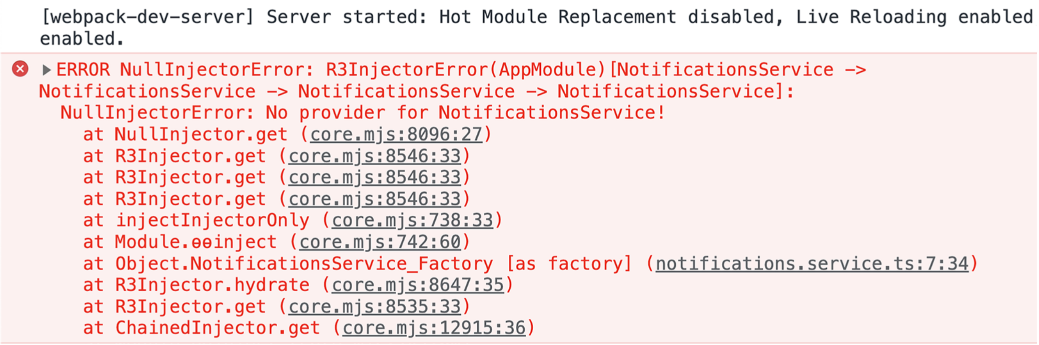
   ```
   ...
   import { NotificationsService } from '../services/notifications.service';

   @NgModule({
     declarations: [HomeComponent, NotificationsManagerComponent],
     imports: [CommonModule, HomeRoutingModule],
     providers: [NotificationsService],
   })
   export class HomeModule {}
   ```

   Boom! We don't have any errors on the console or during compile time. However, we do have the issue of the count not updating in the header. So, how do we alert the developers if they make such a mistake?

5. In order to alert the developer about potential duplicate providers, we will use the `@SkipSelf` decorator from the `@angular/core` package in our `NotificationsService`, and throw an error to notify and modify `NotificationsService`, as follows:

   ```
   import { Injectable, SkipSelf } from '@angular/core';
   import { BehaviorSubject, Observable } from 'rxjs';

   @Injectable({
     providedIn: 'root',
   })
   export class NotificationsService {
     ...
     constructor(@SkipSelf() existingService:
       NotificationsService) {
       if (existingService) {
         throw Error(
           'The service has already been provided in the
            app.
           Avoid providing it again in child  modules'
         );
       }
     }
     ...
   }
   ```

With the previous step now complete, you'll notice that we have a problem, which is that we have failed to provide `NotificationsService` to our app. You should see this in the console:



```
[webpack-dev-server] Server started: Hot Module Replacement disabled, Live Reloading enabled
enabled.
▸ERROR NullInjectorError: R3InjectorError(AppModule)[NotificationsService ->
NotificationsService -> NotificationsService -> NotificationsService]:
  NullInjectorError: No provider for NotificationsService!
    at NullInjector.get (core.mjs:8096:27)
    at R3Injector.get (core.mjs:8546:33)
    at R3Injector.get (core.mjs:8546:33)
    at R3Injector.get (core.mjs:8546:33)
    at injectInjectorOnly (core.mjs:738:33)
    at Module.ɵɵinject (core.mjs:742:60)
    at Object.NotificationsService_Factory [as factory] (notifications.service.ts:7:34)
    at R3Injector.hydrate (core.mjs:8647:35)
    at R3Injector.get (core.mjs:8535:33)
    at ChainedInjector.get (core.mjs:12915:36)
```

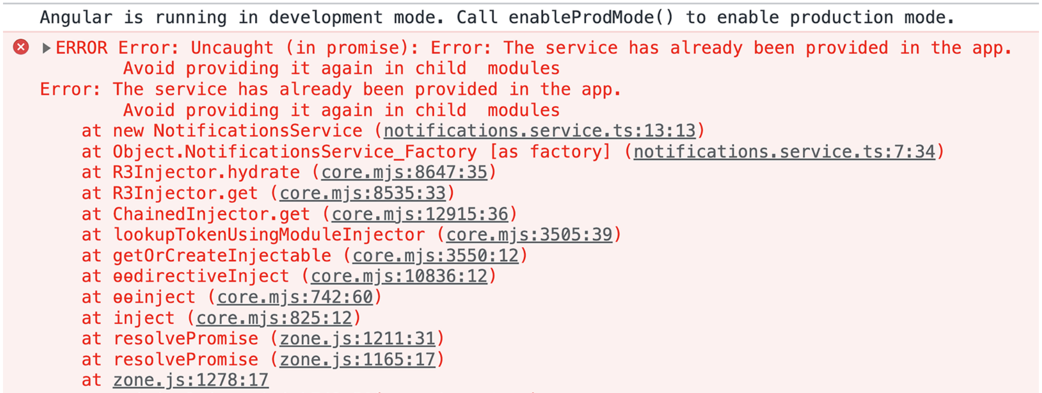
Figure 3.8: An error detailing that NotificationsService can't be injected into NotificationsService

The reason for this is that `NotificationsService` is now a dependency of `NotificationsService` itself. This can't work as it has not already been resolved by Angular. To fix this, we'll also use the `@Optional()` decorator in the next step.

6. All right – now, we'll use the `@Optional()` decorator in `notifications.service.ts,` which is in the constructor for the dependency alongside the `@SkipSelf` decorator. The code should appear as follows:

```
import { Injectable, Optional, SkipSelf } from '@angular/core';
...
export class NotificationsService {
  ...
  constructor(@Optional() @SkipSelf() existingService:
    NotificationsService) {
    if (existingService) {
      throw Error ('The service has already been provided in
        the app. Avoid providing it again in child
        modules');
    }
  }
  ...
}
```

We have now fixed the `NotificationsService` -> `NotificationsService` dependency issue. You should see the proper error for the `NotificationsService` being provided multiple times in the console, as follows:



```
Angular is running in development mode. Call enableProdMode() to enable production mode.
ERROR Error: Uncaught (in promise): Error: The service has already been provided in the app.
        Avoid providing it again in child  modules
Error: The service has already been provided in the app.
        Avoid providing it again in child  modules
    at new NotificationsService (notifications.service.ts:13:13)
    at Object.NotificationsService_Factory [as factory] (notifications.service.ts:7:34)
    at R3Injector.hydrate (core.mjs:8647:35)
    at R3Injector.get (core.mjs:8535:33)
    at ChainedInjector.get (core.mjs:12915:36)
    at lookupTokenUsingModuleInjector (core.mjs:3505:39)
    at getOrCreateInjectable (core.mjs:3550:12)
    at ɵɵdirectiveInject (core.mjs:10836:12)
    at ɵɵinject (core.mjs:742:60)
    at inject (core.mjs:825:12)
    at resolvePromise (zone.js:1211:31)
    at resolvePromise (zone.js:1165:17)
    at zone.js:1278:17
```

Figure 3.9: An error detailing that NotificationsService is already provided in the app

7. Now, we'll safely remove the provided `NotificationsService` from the `providers` array in the `home.module.ts` file, as shown in *step 3*, and check whether the app is working correctly

Bam! We now have a singleton service using the `providedIn` strategy. In the next section, let's discuss how it works.

How it works

Whenever we try to inject a service somewhere, by default, it tries to find a service inside the associated module of where you're injecting the service. When we use `providedIn: 'root'` to declare a service, whenever the service is injected anywhere in the app, Angular knows that it simply must find the service definition in the root scope, instead of trying to find it in the feature modules or anywhere else.

However, you must make sure that the service is only provided once in the entire application. If you provide it in multiple modules, then even with `providedIn: 'root'`, you'll have multiple instances of the service. To avoid providing a service in multiple modules or multiple places in the app, we can use the `@SkipSelf()` decorator with the `@Optional()` decorator in the services' constructor to check whether the service has already been provided in the app.

See also

- Hierarchical DI in Angular (`https://angular.io/guide/hierarchical-dependency-injection`)

Creating a singleton service using forRoot()

In this recipe, you'll learn how to use `ModuleWithProviders` and the `forRoot()` statement to ensure your Angular service is used as a singleton in the entire app. We'll start with an app that has multiple instances of `NotificationsService`, and we'll implement the necessary code to make sure we end up with a single instance of the service in our application.

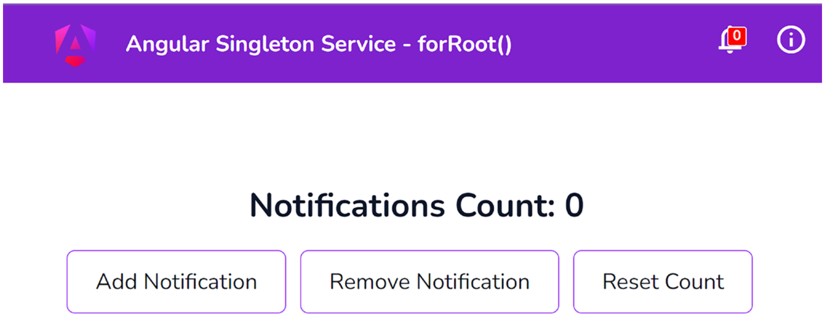
Getting ready

The app that we are going to work with resides in `start/apps/chapter03/ng-singleton-service-forroot`, inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the Terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-singleton-service-forroot
```

This should open the app in a new browser tab, and you should see the following:



Angular Singleton Service - forRoot()

Notifications Count: 0

Figure 3.10: The ng-singleton-service-forroot app running on http://localhost:4200

Now that we have the app running, in the next section, we can move on to the steps for the recipe.

How to do it

In order to make sure you only have a singleton service in the app with the `forRoot` method, you need to understand how `ModuleWithProviders` and the `static forRoot()` method are created and implemented. Perform the following steps:

1. First, we'll make sure that the service has its own module. In many Angular applications, you'll probably see `CoreModule` where the services are provided (given we're not using the `providedIn: 'root'` syntax for some reason). To begin, we'll create a module, named `ServicesModule`, using the following command from the project root:

```
cd start && nx g m services --project ng-singleton-service-forroot
```

2. Let's create a static method `forRoot()` inside the `ServicesModule` class in the `services.module.ts` file. We'll name the method `forRoot` and return a `ModuleWithProviders` object that contains the `NotificationsService`, provided in the `providers` array, as follows:

```
import { ModuleWithProviders, NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { NotificationsService } from './notifications.service';

@NgModule({
  declarations: [],
  imports: [CommonModule],
})
export class ServicesModule {
  static forRoot(): ModuleWithProviders<ServicesModule> {
    return {
      ngModule: ServicesModule,
      providers: [NotificationsService],
    };
  }
}
```

3. Now, we'll remove the `NotificationsService` from the `app.module.ts` file's `providers` array and include `ServicesModule` in the `app.module.ts` file. In particular, we'll add `ServicesModule` in the `imports` array using the `forRoot()` method, as highlighted in the following code block.

   This is because it injects `ServicesModule` with the providers in `AppModule`, for instance, with the `NotificationsService` being provided, as follows:

```
...
import { NotificationsService } from './services/notifications.service';
// <-- Remove the import above
import { ServicesModule } from './services/services.module';
@NgModule({
  declarations: [...],
  imports: [
    ...,
    ServicesModule.forRoot()
  ],
  providers: [
    NotificationsService // <-- Remove this
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

You'll notice that when adding/removing notifications, the count in the header still doesn't change. This is because we're still providing the `NotificationsService` in the `home.module.ts` file.

4. We'll remove the `NotificationsService` from the `providers` array in the `home.module.ts` file, as follows:

```
...
import { NotificationsService } from '../services/notifications.service';
//  <-- Remove the above import
import { ServicesModule } from '../services/services.module';

@NgModule({
  declarations: [HomeComponent,
    NotificationsManagerComponent],
  imports: [CommonModule, HomeRoutingModule,
    ServicesModule],
  providers: [
    NotificationsService // <-- Remove this
  ],
})
export class HomeModule {}
```

Great job. Now that we have finished the recipe, in the next section, let's discuss how it works.

How it works

`ModuleWithProviders` acts as a wrapper for an `NgModule`, bundling it with a `providers` array. It's used to configure an `NgModule` along with its providers, ensuring that when the module is imported elsewhere, it brings along its providers, too. In our `ServicesModule`, we've created a `forRoot` method that returns a `ModuleWithProviders`. It includes our `NotificationsService`, which allows us to have a single instance of this service throughout the app, avoiding multiple instances that would normally occur if `NotificationsService` was provided in the `providers` array of `ServicesModule` and imported into various modules. Therefore, to ensure a single instance, `ServicesModule` should be imported using the `ModuleWithProviders` approach, not in the standard way. That is why when using the `ModuleWithProviders` approach, we don't import the `ServicesModule` in the regular way, as follows:

```
@NgModule({
  ...
  imports: [..., ServicesModule],
})
```

Instead, we import it using the `forRoot` method which ensures the `NotificationService` is only provided once in the entire app as follows:

```
@NgModule({
...
  imports: [..., ServicesModule.forRoot()],
})
```

Now that you understand how the recipe works, see the next section for some useful links.

See also

- The `ModuleWithProviders` Angular documentation (`https://angular.io/api/core/ModuleWithProviders`)
- The `ModuleWithProviders` migration documentation (`https://angular.io/guide/migration-module-with-providers%20`)

Providing alternate classes against the same DI Token

In this recipe, you'll learn how to provide two different services to the app using **Aliased class providers**. This is extremely helpful in complex applications where you need to narrow down the implementation of the service/class for some components/modules, that is, providing different classes against the same DI token to have a polymorphic behavior. Additionally, aliasing is used in component/service unit tests to mock the dependent service's actual implementation so that we don't rely on it.

Getting ready

The app that we are going to work with resides in `start/apps/chapter03/ng-aliased-class-providers`, inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the Terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve ng-aliased-class-providers
   ```

   This should open the app in a new browser tab, and you should see the app as shown in *Figure 3.11*.
3. Click on the **Login as Admin** button. You should see something similar to the following screenshot:

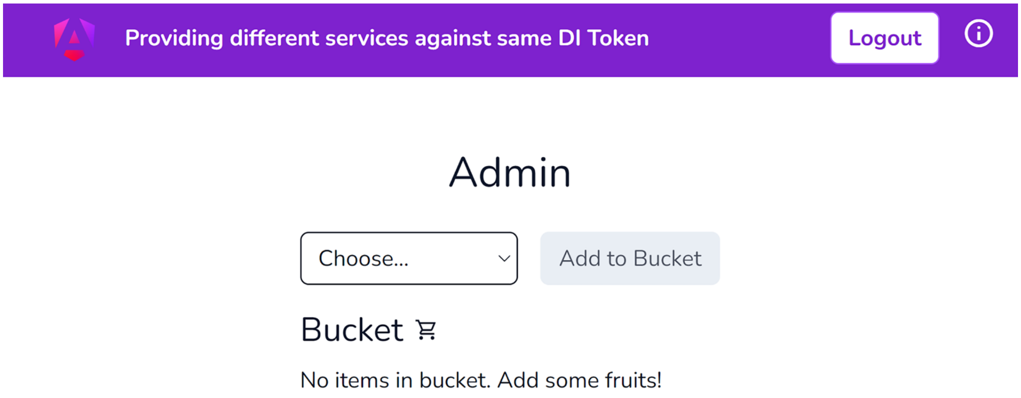

Providing different services against same DI Token
Logout
Admin
Choose...
Add to Bucket
Bucket
No items in bucket. Add some fruits!

Figure 3.11: The ng-aliased-class-providers app running on http://localhost:4200

Now that we have the app running, let's move to the next section to follow the steps for the recipe.

How to do it

We have a standalone component named `BucketComponent`, which is being used in both the admin and employee components. `BucketComponent` uses `BucketService` behind the scenes to add/remove items from and to a bucket. For the employee, we'll restrict the ability to remove an item by providing an `aliased` class provider and a replacement of `BucketService` named `EmployeeBucketService`. This is so that we can override the remove item functionality. Perform the following steps to get started:

1. We'll start by creating `EmployeeBucketService` within the `employee` folder. Run the following command from the workspace root:

   ```
   cd start && nx g service employee/employee-bucket --project ng-aliased-class-providers
   ```

2. Next, we'll extend `EmployeeBucketService` from `BucketService` so that we get all the goodness of the `BucketService` class. Let's modify the code as follows:

   ```
   import { Injectable } from '@angular/core';
   import { BucketService } from '../bucket/bucket.service';
   ...
   export class EmployeeBucketService extends BucketService {
     constructor() {
       super();
     }
   }
   ```

3. We will now override the `removeItem` method to display a simple `alert`, mentioning that the employees can't remove items from the bucket. Your code should appear as follows:

   ```
   ...
   export class EmployeeBucketService extends BucketService {
     constructor() {...}
     override removeItem() {
       alert('Employees can not delete items');
     }
   }
   ```

4. As a final step, we need to provide the `aliased` class provider to the `employee.component.ts` file, as follows:

   ```
   ...
   import { BucketService } from '../bucket/bucket.service';
   import { EmployeeBucketService } from './employee-bucket.service';

   @Component({
     ...
   ```

```
    providers: [{
      provide: BucketService,
      useClass: EmployeeBucketService,
    }],
  })
  export class EmployeeComponent {}
```

If you now log in as an employee in the app and try to remove an item, you'll see an alert pop up, which says `Employees cannot delete items.`

How it works

When we inject a service into a component, Angular tries to find that component inside the component/module we've provided the dependency for and then by moving up the hierarchy of components and modules. Our `BucketService` is provided in `'root'`, using the `providedIn: 'root'` syntax. Therefore, it resides at the top of the hierarchy. However, since, in this recipe, we use an `aliased` class provider in the `EmployeeComponent` class against the DI token `BucketService`, when Angular searches for `BucketService` for the `EmployeeComponent`, it quickly finds the `EmployeeBucketService` against the token in the `EmployeeComponent` and stops there – i.e., it doesn't reach the "root" to get the actual `BucketService`. This is what we intended.

See also

- DI in Angular (`https://angular.io/guide/dependency-injection`)
- Hierarchical injectors in Angular (`https://angular.io/guide/hierarchical-dependency-injection`)

Dynamic configurations using value providers

In this recipe, you'll learn how to use value providers in Angular to provide constants and config values to your app. We'll start with the same example from the previous recipe, which involves `EmployeeComponent` and `AdminComponent` using the `BucketComponent` to manage a bucket of fruits. We will restrict the `EmployeeComponent` from deleting items from the bucket by using the configuration with a value provider. As a result, the employees won't even see the **Delete** button.

Getting ready

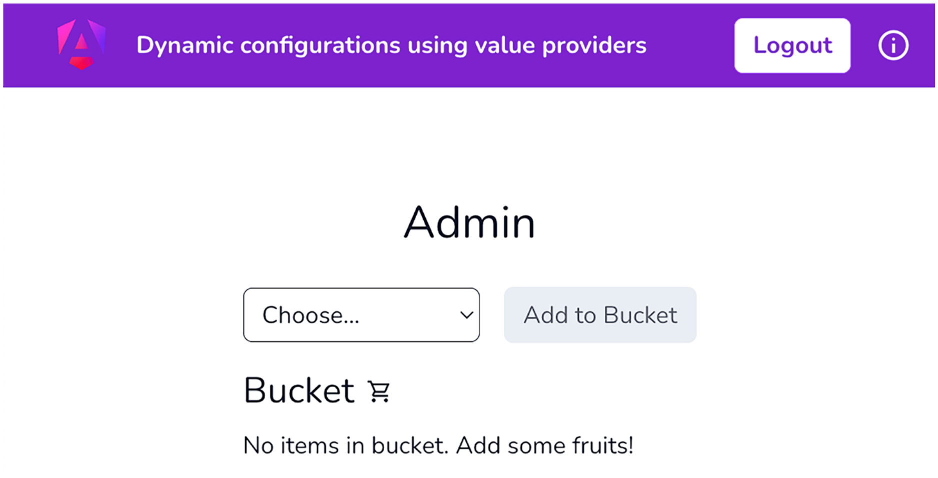
The app that we are going to work with resides in `start/apps/chapter03/ng-value-providers` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve ng-value-providers
   ```

   This should open the app in a new browser tab, and you should see the app as shown in *Figure 3.12*.

3. Click on the **Login as Admin** button. You should see something like the following screenshot:



No items in bucket. Add some fruits!

Figure 3.12: The ng-value-providers app running on http://localhost:4200

Now that you see the app running, let's look at the next section to follow the recipe.

How to do it

We have a standalone component named `BucketComponent`, which is being used in both the admin and employee components. `BucketComponent` uses `BucketService` behind the scenes to add/remove items from and to a bucket. For the employee, we'll restrict the ability to remove an item by providing a value provider. This is so that we can override the remove item functionality. Let's begin with the following steps:

1. First, we'll start by creating the value provider with `InjectionToken` within a new file, named `app-config.ts`, in the root of the project. The code should appear as follows:

```
import { InjectionToken } from '@angular/core';
export interface IAppConfig {
  canDeleteItems: boolean;
}
export const APP_CONFIG = new InjectionToken<IAppConfig>('APP_CONFIG');
export const AppConfig: IAppConfig = {
  canDeleteItems: true,
};
```

   Before we can actually use this `AppConfig` constant in our `BucketComponent`, we need to register it to the `AppModule` so that when we inject this in the `BucketComponent`, the value of the provider is resolved.

2. Let's add the provider to the `app.module.ts` file, as follows:

```
...
import { AppConfig, APP_CONFIG } from './app-config';

@NgModule({
  declarations: [AppComponent],
  imports: [...],
  providers: [{
    provide: APP_CONFIG,
    useValue: AppConfig,
  }],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

   Now, the app knows about the `AppConfig` constants. The next step is to use this constant in `BucketComponent`.

3. We'll use the `inject` method to inject it inside the `BucketComponent` class, in the `bucket/bucket.component.ts` file, as follows:

```
import { Component, inject, OnInit } from '@angular/core';
...
import { APP_CONFIG } from '../app-config';
...
export class BucketComponent implements OnInit {
  bucketService = inject(BucketService);
  appConfig = inject(APP_CONFIG);
  ...
}
```

   Great! The constant has been injected. Now, if you refresh the app, you shouldn't get any errors. The next step is to use the `canDeleteItems` property from `config` in `BucketComponent` to show/hide the `delete` button.

4. Now, we'll add an `*ngIf` directive in the `bucket/ bucket.component.html` file to only show the `delete` button if the value of `appConfig.canDeleteItems` is `true`. Update the element with the `fruites__item__delete-icon` class, as follows:

```
...
<div *ngIf="appConfig.canDeleteItems"
  class="fruites__item__delete-icon"
  (click)="deleteFromBucket(item)">
  <div class="material-symbols-outlined">delete</div>
</div>
...
```

You can test whether everything works by setting the `AppConfig` constant's `canDeleteItems` property to `false`. Note that the **Delete** button is now hidden for both the admin and employee. Once tested, set the value of `canDeleteItems` back to `true` again.

Now, we have everything set up. Let's add a new constant so that we can hide the **Delete** button for the employee only.

5. Let's create an employee configuration object now. We'll create an `employee.config.ts` file inside the `employee` folder, and we will add the following code to it:

```
import { IAppConfig } from '../app-config';
export const EmployeeConfig: IAppConfig = {
  canDeleteItems: false,
};
```

6. Now, we'll provide this `EmployeeConfig` constant to the `EmployeeComponent` for the same `APP_CONFIG` injection token. The code in the `employee.component.ts` file should look as follows:

```
...
import { APP_CONFIG } from '../app-config';
import { EmployeeConfig } from './employee.config';

@Component({
  ...
  providers: [{
    provide: APP_CONFIG,
    useValue: EmployeeConfig,
  }],
})
export class EmployeeComponent {}
```

And we're done! The recipe is now complete. You can see that the **Delete** button is visible to the admin but hidden for the employee. It's all thanks to the magic of value providers.

How it works

When we inject a token into a component, Angular tries to find the resolved value of the token in the injected place and then by moving up the hierarchy of components and modules. We provided the `EmployeeConfig` object against the `APP_CONFIG` token in the `EmployeeComponent` class. When Angular tries to resolve the token's value for `BucketComponent`, it finds it early as `EmployeeConfig`, within the `EmployeeComponent` instead of the value provided in the `AppModule` as `AppConfig`. Therefore, Angular stops right there and doesn't reach `AppModule`. This is amazing, as we can now have a global configuration and override the configuration within nested modules/components.

See also

- DI in Angular (`https://angular.io/guide/dependency-injection`)
- Hierarchical injectors in Angular (`https://angular.io/guide/hierarchical-dependency-injection`)

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

4

Understanding Angular Animations

In this chapter, you'll learn about working with **animations** in Angular. You'll learn about multi-state animations, staggering animations, and keyframe animations, as well as how to implement animations for switching routes in your Angular apps and how to conditionally disable animations.

The following are the recipes that we're going to cover in this chapter:

- Creating your first two-state Angular animation
- Working with multi-state animations
- Creating complex Angular animations using keyframes
- Animating lists in Angular using stagger animations
- Sequential vs parallel animations in Angular
- Route animations in Angular
- Disabling Angular animations conditionally

Technical requirements

For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter04`.

Creating your first two-state Angular animation

In this recipe, you'll create a basic two-state Angular animation that has a fading effect. We'll start with an Angular app with a UI already built into it. We'll then enable animations in the app using Angular animations and will move toward creating our first animation.

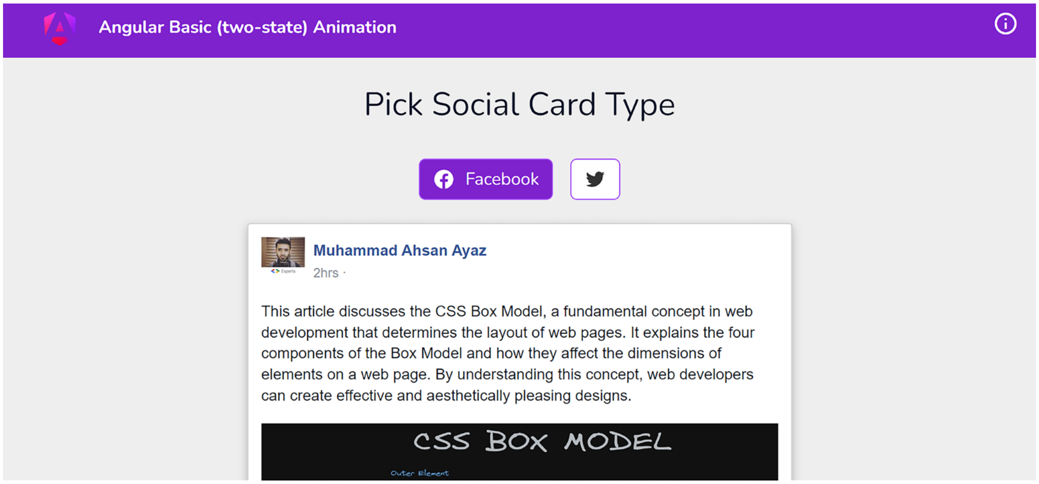
Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-basic-animation` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-basic-animation
```

This should open the app in a new browser tab, and you should see the following:



Angular Basic (two-state) Animation

Pick Social Card Type

Muhammad Ahsan Ayaz

2hrs ·

This article discusses the CSS Box Model, a fundamental concept in web development that determines the layout of web pages. It explains the four components of the Box Model and how they affect the dimensions of elements on a web page. By understanding this concept, web developers can create effective and aesthetically pleasing designs.

Figure 4.1: ng-basic-animation app running on http://localhost:4200

Now that we have the app running, we will move on to the steps for the recipe.

How to do it…

We have an app that doesn't have Angular animations configured at all. We will create a fading effect for the cards using Angular animations. Let's continue with the steps as follows:

1. First, we'll import the `provideAnimations` function from the `@angular/platform-browser/animations` package in our `src/app/app.config.ts` file so we can use animations in the app. We'll use it in the `providers` array as follows:

```
...
import { provideAnimations } from '@angular/platform-browser/animations';
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(appRoutes, withEnabledBlockingInitialNavigation()),
    provideAnimations()
  ],
};
```

2. Now modify the `app.component.ts` file to add the animations as follows:

```
...
import { trigger, transition, style, animate } from '@angular/
animations';
@Component({
  ...
  imports: [CommonModule, FbCardComponent,
    TwitterCardComponent],
  animations: [
    trigger('fadeInOut', [
      transition(':enter', [
        style({ opacity: 0, scale: 0.85 }),
        animate('200ms 100ms', style({ opacity: 1,
          scale: 1 })),
      ]),
      transition(':leave', [
        style({ opacity: 1, scale: 1 }),
        animate('100ms', style({ opacity: 0, scale: 0.85 })),
      ]),
    ]),
  ],
})
...
```

3. Finally, add the `fadeInOut` animation on both the cards in the `app.component.html` file as follows:

```
  <!-- Toolbar -->
<div class="toolbar" role="banner">...</div>
<main class="content" role="main">
  <div class="type-picker mb-8">...</div>
  <ng-container [ngSwitch]="selectedCardType">
    <app-fb-card [@fadeInOut]
      *ngSwitchCase="'facebook'"></app-fb-card>
    <app-twitter-card [@fadeInOut]
      *ngSwitchCase="'twitter'"></app-twitter-card>
  </ng-container>
</main>}
```

Great! You have now implemented a basic *fade-in* <=> *fade-out* animation for the cards. Simple, but pretty! Refer to the next section to understand how the recipe works.

How it works...

Angular provides its own animation API that allows you to animate any property that the CSS transitions work on. The benefit is that you can configure them dynamically based on the conditions required. If we were to create the same behavior in CSS, we would have to do the following:

1. We would have to create the keyframes in CSS as follows:

   ```
   @keyframes fadeIn {
     0% { opacity: 0; transform: scale(0.85); }
     100% { opacity: 1; transform: scale(1); }
   }

   @keyframes fadeOut {
     0% { opacity: 1; transform: scale(1); }
     100% { opacity: 0; transform: scale(0.85); }
   }
   ```

   Creating the CSS classes to apply these animations:

   ```
   /* For elements that are entering */
   .fade-in {
     animation: fadeIn 200ms 100ms forwards;
   }

   /* For elements that are leaving */
   .fade-out {
     animation: fadeOut 100ms forwards;
   }
   ```

2. Then we would have to add and remove the CSS classes on each element as they are **created** on the DOM and are **removed**. However, Angular handles this process using the built-in `:enter` and `:leave` states, which get triggered once an item is added or removed from the DOM, respectively.

Even with the steps above, more challenges can appear when working with such animations. Thanks to Angular animations, we can implement these a lot faster.

We first used the `trigger` function to register the animation named `fadeInOut`. Then we had the `:enter` and `:leave` transitions being registered with the `transition` function. Finally, we defined the styles and animations for those transitions using the `style` and `animate` functions. Note that we're using `'200ms 100ms..'` in the `:enter` transition. `200ms` is the duration of the transition while `100ms` is the delay. We add this `delay` so we can wait for the `:leave` transition of the previously shown card to finish before we can move to the `:enter` transition of the next card to be shown. Let's delve deeper into each of the functions we used:

1. The `trigger` function: The `trigger` function is used to define an animation trigger in Angular. The first argument is the name of the trigger, which will be used in the template to bind the animation to a specific element. The second argument is an array of state and transition definitions. For example, `trigger('fadeInOut', [...])` registers an animation trigger named `'fadeInOut'`.
2. The `:enter` and `:leave` transitions: `:enter` is an alias for the `void => *` state transition. It represents the state of an element that is being added to the DOM. `:leave` is an alias for the `* => void` state transition. It represents the state of an element that is being removed from the DOM. These aliases are useful for animations that are common to elements entering or leaving the view, such as fade-in and fade-out animations.
3. The `transition` function: The `transition` function is used to define the states between which transitions will occur. It accepts two arguments: the first is a string that defines the state change expression, and the second is an array of animation steps to run when the transition is triggered. For example, `transition(':enter', [...])` defines the animation steps that will be executed when an element enters the view.
4. `style`: The `style` function is used to define a set of CSS styles that will be used in the animation. It accepts an object where the keys are CSS properties and the values are the desired values for those properties. For example, `style({ opacity: 0, scale: 0.85 })` sets the opacity to `0` and scales the element down to 85% of its original size.
5. `animate`: The `animate` function is used to define the timing and easing for the transition between styles. The first argument is a string that defines the duration, delay, and easing curve. For instance, `200ms 100ms` means the animation will last 200 ms and will start after a delay of 100 ms. The second argument is the style or a set of styles that the animation will transition to. For example, `animate('200ms 100ms', style({ opacity: 1, scale: 1 }))` will transition the element to full opacity and original scale over 300 ms, after waiting for 100 ms.

See also

- Animations in Angular: `https://angular.io/guide/animations`
- *Angular Animations Explained with Examples*: `https://www.freecodecamp.org/news/angular-animations-explained-with-examples/`

Working with multi-state animations

In this recipe, we'll work with Angular animations containing multiple states. This means that we'll work with more than two states for a particular item. We'll be using the same Facebook and Twitter cards example for this recipe as well.

We'll configure the following states for both cards:

- The state when a card appears on the screen.
- The state when the user hovers over a card.
- The state when the user moves the mouse away from the card.

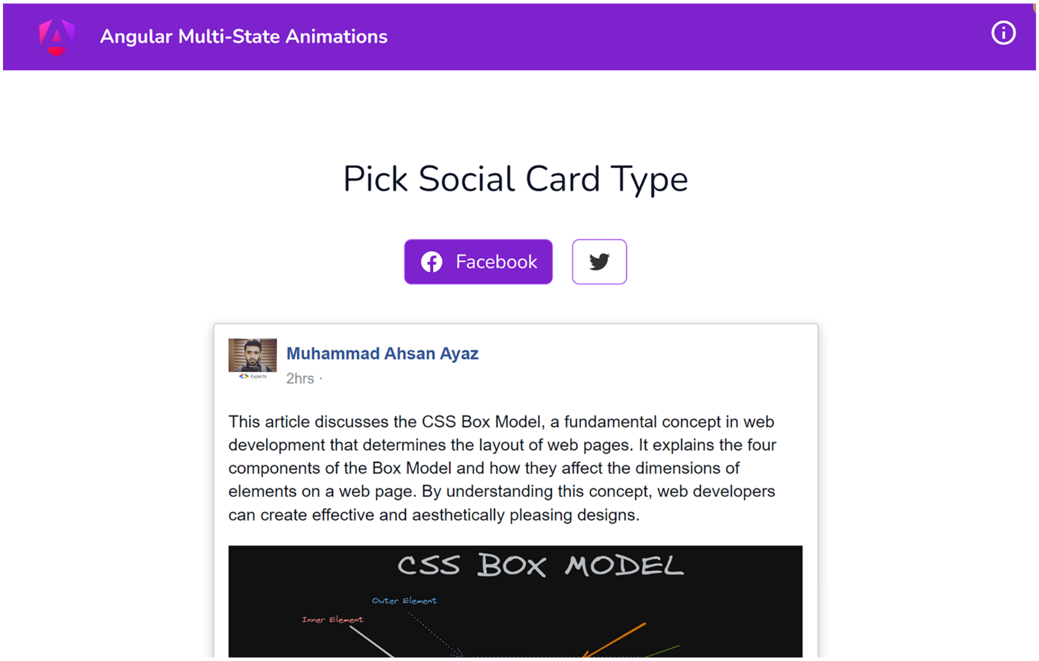
Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-multi-state-animations` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-multi-state-animations
```

This should open the app in a new browser tab, and you should see the following:



This article discusses the CSS Box Model, a fundamental concept in web development that determines the layout of web pages. It explains the four components of the Box Model and how they affect the dimensions of elements on a web page. By understanding this concept, web developers can create effective and aesthetically pleasing designs.

Figure 4.2: ng-multi-state-animations app running on http://localhost:4200

Now that we have the app running locally, let's look at the steps of the recipe in the next section.

How to do it...

We already have a working app that has a single animation built for the reach of social cards. When you tap either the Facebook or Twitter button, you'll see the respective card appearing with a slide-in animation from left to right. To keep the recipe simple, we'll implement two more states and two animations for when the user moves the mouse cursor over the card and when the user moves away from the card. Let's add the relevant code in the following steps:

1. We'll start by adding two `@HostListener` instances to `FbCardComponent` in the `components/fb-card/fb-card.component.ts` file, one for the `mouseenter` event on the card and one for the `mouseleave` event. We'll name the states `hovered` and `active`, respectively. The code should look as follows:

```
import { Component, HostListener } from '@angular/core';
...
@Component({...})
export class FbCardComponent {
  cardState: 'active' | 'hovered' = 'active';
  @HostListener('mouseenter')
  onMouseEnter() {
    this.cardState = 'hovered';
  }

  @HostListener('mouseleave')
  onMouseLeave() {
    this.cardState = 'active';
  }
}
```

2. Now, we'll do the same for `TwitterCardComponent` in the `components/twitter-card/twitter-card-component.ts` file. The code should look as follows:

```
import { Component, HostListener } from '@angular/core';
...
@Component({...})
export class TwitterCardComponent {
  cardState: 'active' | 'hovered' = 'active';
  @HostListener('mouseenter')
  onMouseEnter() {
    this.cardState = 'hovered';
  }

  @HostListener('mouseleave')
  onMouseLeave() {
    this.cardState = 'active';
  }
}
```

There should be no visual change so far since we're only updating the `cardState` variable to have the hover and active states. We haven't defined the transitions yet for the animation.

3. We'll now define our state for when the user's cursor enters the card, that is, the `mouseenter` event. The state is called `hovered` and should look as follows in the `animation.ts` file:

```
...
export const cardAnimation = trigger('cardAnimation', [
  state('active', style({
    color: 'rgb(51, 51, 51)',
    backgroundColor: 'white'
  })),
  state('hovered', style({
    transform: 'scale3d(1.05, 1.05, 1.05)',
    backgroundColor: '#333',
    color: 'white'
  })),
  transition('void => active', [...]),
])
```

If you refresh the app now, tap either the Facebook or Twitter button, and hover the mouse over the card, you'll see the card's UI changing. That's because we changed the state to `hovered`. However, there's no animation yet between the style changes. Let's add the animation in the next step.

4. We'll add the `active => hovered` transition now in the `animations.ts` file so that we can smoothly navigate from `active` to the `hovered` state:

```
...
export const cardAnimation = trigger('cardAnimation', [
  state('active', style(...)),
  state('hovered', style(...)),
  transition('void => active', [...]),
  transition('active => hovered', [
    animate('0.3s 0s ease-out', style({
      transform: 'scale3d(1.05, 1.05, 1.05)',
      backgroundColor: '#333',
      color: 'white'
    }))
  ]),
])
```

You should now see the smooth transition on the `mouseenter` event if you refresh the app.

5. Finally, we'll add the final transition, `hovered => active`, so when the user leaves the card, we revert to the active state with a smooth animation. The code should look as follows:

```
...
export const cardAnimation = trigger('cardAnimation', [
  state('active', style(...)),
  state('hovered', style(...)),
  transition('void => active', [...]),
  transition('active => hovered', [...]),
  transition('hovered => active', [
    animate('0.3s 0s ease-out', style({
      transform: 'scale3d(1, 1, 1)',
      color: 'rgb(51, 51, 51)',
      backgroundColor: 'white'
    }))
  ]),
])
```

Ta-da! You now know how to implement different states and different animations on a single element using **Angular animations**.

How it works...

Angular uses triggers to understand what state the animation is in. An example syntax looks as follows:

```
<div [@animationTriggerName]="expression">...</div>;
```

The `expression` can be a valid JavaScript expression, and it evaluates to the name of the state. In our case, we bind it to the `cardState` property, which contains either `active` or `hovered`. Therefore, we end up with three transitions for our cards:

- `void => active` (when the element is added to the DOM and is rendered)
- `active => hovered` (when the `mouseenter` event triggers on the card)
- `hovered => active` (when the `mouseleave` event triggers on the card)

See also

- Triggering the animation: `https://angular.io/guide/animations#triggering-the-animation`
- Reusable animations: `https://angular.io/guide/reusable-animations`

Creating complex Angular animations using keyframes

Since you already know about Angular animations from the previous recipes, you might be thinking, "That's easy enough." Well, time to level up your animation skills in this recipe. You'll create a complex Angular animation using `keyframes` in this recipe to get started with writing some advanced animations.

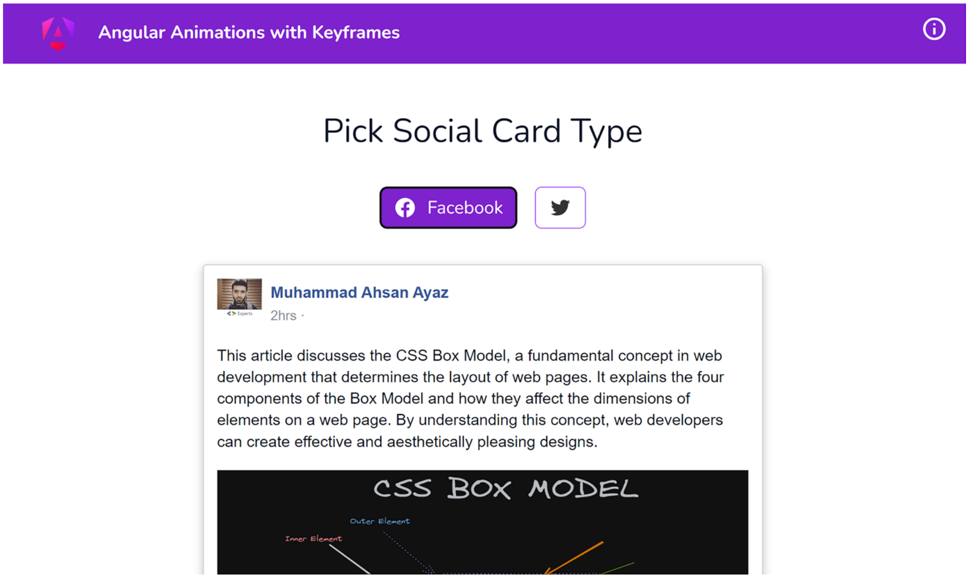
Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-animations-keyframes` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-animations-keyframes
```

This should open the app in a new browser tab, and you should see the following:



This article discusses the CSS Box Model, a fundamental concept in web development that determines the layout of web pages. It explains the four components of the Box Model and how they affect the dimensions of elements on a web page. By understanding this concept, web developers can create effective and aesthetically pleasing designs.

Figure 4.3: ng-animations-keyframes app running on http://localhost:4200

Now that we have the app running locally, let's look at the steps of the recipe in the next section.

How to do it...

We have an app right now that has a single transition, that is, `void => active`, which triggers when the element enters the DOM. Right now, the animation is pretty simple. We'll use the `keyframes` function to build a complex animation:

1. Let's begin with adding the `keyframes` function from `@angular/animations` to the `animations.ts` file as follows:

```
import {
  ...,
  keyframes
} from '@angular/animations';
...
```

2. Now, we'll convert the single-style animation for the `void => transition` to use keyframes as follows:

```
...
export const cardAnimation = trigger('cardAnimation', [
  ...,
  transition('void => *', [
    style({ // ← Remove this style
      transform: 'translateX(-200px)',
      opacity: 0
    }),
    animate('0.2s ease', keyframes([
      style({
        transform: 'translateX(-200px)',
        offset: 0
      }),
      style({
        transform: 'translateX(0)',
        offset: 1
      })
    ]))
  ]),
])
```

Notice that previously we had to define the initial style and the `animate` function. Now we can define the same inside the `keyframes` function in chronological order. If you refresh the app now and try it, you'll still see the same animation as before. But now we have it using `keyframes`.

3. Finally, let's start adding some complex animations. Let's start the animation with a scaled-down card by adding `scale3d` to the `transform` property of `style` at `offset: 0`. We'll also increase the animation time to `1.5s`:

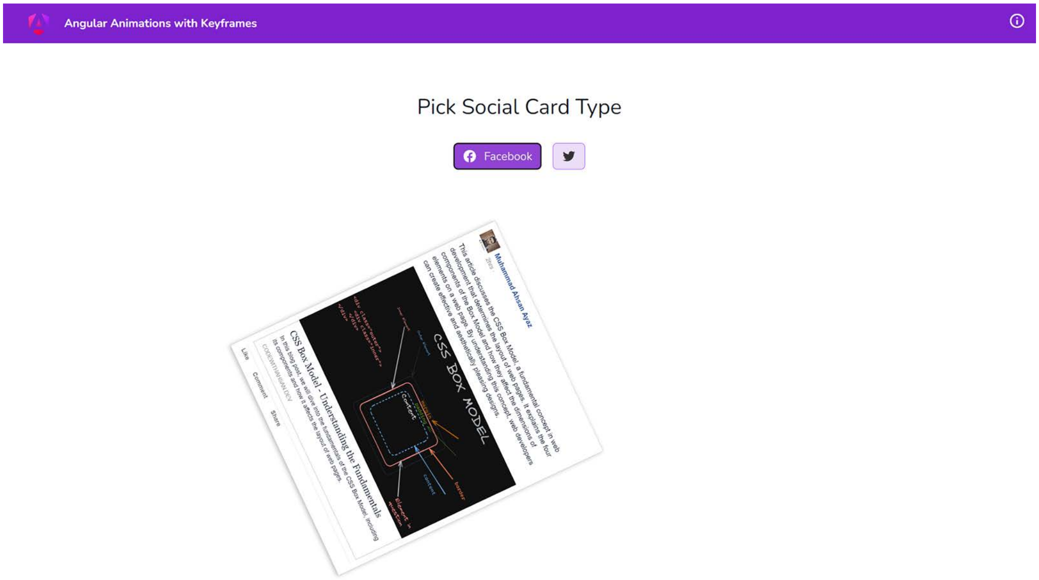
```
...
export const cardAnimation = trigger('cardAnimation', [
  transition('void => active', [
    animate('1.5s ease', keyframes([
      style({
        transform: 'translateX(-200px)
          scale3d(0.4,0.4,0.4)',
        offset: 0
      }),
      style({...})
    ]))
  ]),
])
```

You should now see that the card animation starts with a small card that slides from the left and moves toward the right, increasing in size.

4. Now we'll implement a *zig-zag-ish* animation for the appearance of the card instead of the slide-in animation. Let's add the following keyframe elements to the `keyframes` array to add a bumpy effect to our animation:

```
...
export const cardAnimation = trigger('cardAnimation', [
  transition('void => *', [
    animate('1.5s 0s ease', keyframes([
      style({
        transform: 'translateX(-200px)
          scale3d(0.4,0.4,0.4)',
        offset: 0
      }),
      style({
        transform: 'translateX(0px) rotate(-90deg)
        scale3d(0.5, 0.5, 0.5)',
        offset: 0.25
      }),
      style({
        transform: 'translateX(-200px) rotate(90deg)
        translateY(0) scale3d(0.6, 0.6, 0.6)',
        offset: 0.5
      }),
      style({
        transform: 'translateX(0)',
        offset: 1
      })
    ]))
  ]),
])
```

If you refresh the app and tap any of the buttons, you should see the card bumping to the right wall, and then to the left wall of the card, before returning to the normal state:



Angular Animations with Keyframes

Pick Social Card Type

Figure 4.4: Card bumping to the right and then the left wall

5. As the last step, we'll spin the card clockwise before it returns to its original position. For that, we'll use `offset: 0.75`, using the `rotate` function with some additional angles. The code should look as follows:

```
...
export const cardAnimation = trigger('cardAnimation', [
  transition('void => *', [
    animate('1.5s 0s ease', keyframes([
      style({...}),
      style({...}),
      style({...}),
      style({
        transform: 'translateX(-100px) rotate(135deg)
        translateY(0) scale3d(0.6, 0.6, 0.6)',
        offset: 0.75
      }),
      style({...})
    ]))
  ]),
])
```

Awesome! You now know how to implement complex animations in Angular using the `keyframes` function. You will see in the next section how it works.

How it works...

For complex animations in Angular, the `keyframes` function from the `@angular/animations` package is an amazing way of providing different timing offsets for the animation throughout its journey. We can define the offsets using the `style` function, which returns an object of type `AnimationStyleMetadata`. The `style` function takes tokens as inputs, which are a key-value pair where the keys are of the string type and the value can be either a string or a number. Essentially, a token represents a CSS property. This allows us to pass the `offset` property as shown in the recipe, which can have a value between `0` and `1`, reflecting the timing from `0%` to `100%` of the animation. Thus, we can define different styles for different offsets to create advanced animations.

See also

- Animations in Angular: `https://angular.io/guide/animations`
- Angular Animations Explained with Examples: `https://www.freecodecamp.org/news/angular-animations-explained-with-examples/`

Animating lists in Angular using stagger animations

No matter what kind of web application you build today, you are likely to implement some sort of list in it. And to make those lists even better, why not implement elegant animations for them? In this recipe, you'll learn how to animate lists in Angular using stagger animations.

Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-animating-lists` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-animating-lists
```

This should open the app in a new browser tab, and you should see the following:

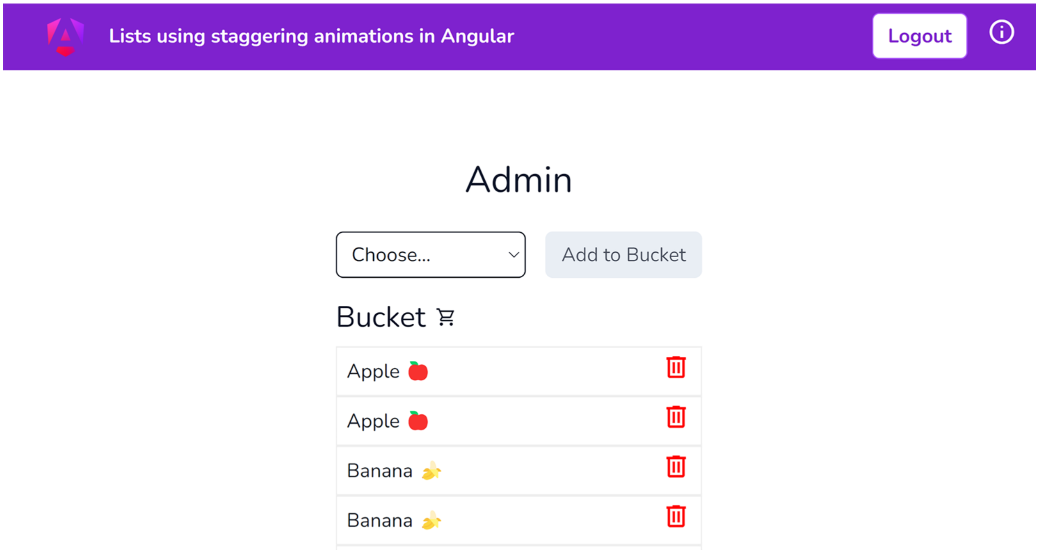

Lists using staggering animations in Angular
Logout
Admin
Choose...
Add to Bucket
Bucket
Apple
Apple
Banana
Banana

Figure 4.5: ng-animating-lists app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

We have an app right now that has a list of bucket items. We need to animate the list using staggering animations. We'll be doing this step by step. I'm excited – are you?

Cool. We'll go through the following steps for the recipe:

1. First, let's provide Angular Animations using the `provideAnimations` function from Angular in the `src/app/app.config.ts` file as follows:

```
...
import { provideAnimations } from '@angular/platform-browser/animations';
import { appRoutes } from './app.routes';
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(appRoutes,
      withEnabledBlockingInitialNavigation()),
    provideAnimations()
  ],
```

2. Now, create a file named `animations.ts` in the project's `app` folder and add the following code to register a basic list item animation named `listItemAnimation`:

```
import { trigger, style, animate, transition } from '@angular/
animations';
export const ANIMATIONS = {
  LIST_ITEM_ANIMATION: trigger('listItemAnimation', [
    transition(':enter', [
      style({ opacity: 0 }),
      animate('0.5s ease', style({ opacity: 1 })),
    ]),
    transition(':leave', [
      style({ opacity: 1 }),
      animate('0.5s ease', style({ opacity: 0 })),
    ]),
  ]),
};
```

3. Now, we'll add the animation to `BucketComponent` in the `app/bucket/bucket.component.ts` file as follows:

```
...
import { ANIMATIONS } from '../../../constants/animations';
@Component({
  ...
  animations: [ANIMATIONS.LIST_ITEM_ANIMATION]
})
```

   Since we have the animation imported into the component, we can use it in the template now.

4. Let's add the animation to the `html` element, with the class `fruits__item` in the `bucket.component.html` file, as follows:

```
<div class="fruits__item" *ngFor="let item of bucket"
  @listItemAnimation>
  ...
</div>
```

   If you now refresh the app and add an item to the bucket list, you should see it appear with a fade-in effect. And if you delete an item, you should see it disappear with the animation as well.

5. We'll modify `LIST_ITEM_ANIMATION` now to use the `stagger` function. This is because the staggering animation is applied on the list, and not the list items. First, we need to import the `stagger` function from `@angular/animations`. Then we need to remove everything from the trigger array, and then create a wildcard transition for the list as follows:

```
import {
  ...,
  stagger,
} from '@angular/animations';
export const ANIMATIONS = {
  LIST_ITEM_ANIMATION: trigger('listItemAnimation', [
    transition('* <=> *', [
      // we'll add more code here
    ]),
  ]),
};
```

6. Now we'll add a query for what happens when a new item is added to the list. We'll use a staggering animation here. The code should look as follows:

```
import { trigger, style, animate, transition, stagger, query } from '@
angular/animations';
export const ANIMATIONS = {
  LIST_ITEM_ANIMATION: trigger('listItemAnimation', [
    transition('* <=> *', [
      query(
        ':enter',
        [
          style({ opacity: 0 }),
          stagger(100, [
            animate('0.5s ease', style({ opacity: 1 }))
          ]),
        ],
        { optional: true }
      ),
    ]),
  ]),
};
```

7. Now we'll add the query for when an item leaves the list. The code should look as follows:

```
export const ANIMATIONS = {
  LIST_ITEM_ANIMATION: trigger('listItemAnimation', [
    transition('* <=> *', [
      query(':enter', [...],
        { optional: true }
      ),
      query(
        ':leave',
        [
          style({ opacity: 1 }),
          animate('0.5s ease', style({ opacity: 0 }))
        ],
        { optional: true }
      ),
    ]),
  ]),
};
```

8. Now we can apply the animation to the list itself. Update the `bucket.component.html` as follows to put the animation on the `div` with the class `fruits` instead:

```
  ...
  <div class="fruits" *ngIf="$bucket | async as bucket"
    [@listItemAnimation]="bucket.length">
    ...
</div>
...
```

Notice that we're binding the `[@ listAnimationlistItemAnimation]` property to `bucket.length`. This will make sure that the animation triggers whenever the length of the bucket changes, that is, when an item is added or removed from the bucket. And this is possible because of the transition `('* <=> *')`.

Awesome! You now know how to implement staggering animations for lists in Angular. You'll see in the next section how it works.

How it works...

Stagger animations only work inside `query` functions and are applied on the list (containing the items) instead of the items themselves. To search or query the items, we first use the `query` function. Then we use the `stagger` function to define how many milliseconds of staggering we want before the animation starts for the next list item. We also use the animation in the `stagger` functions to define the animation for each element found in the query. Notice that we're using `{ optional: true }` for both the `:enter` query and the `:leave` query. We're using it because if there is no item to animate, either when the app starts or if all items are removed, Angular throws an error because it can't find anything to animate.

See also

- Animations in Angular: `https://angular.io/guide/animations`
- Angular animations stagger docs: `https://angular.io/api/animations/stagger`

Sequential vs parallel animations in Angular

In this recipe, you'll learn how to run Angular animations in a sequence vs in parallel. This is handy for when we want to have one animation finished before we start the next one, or to run the animations simultaneously.

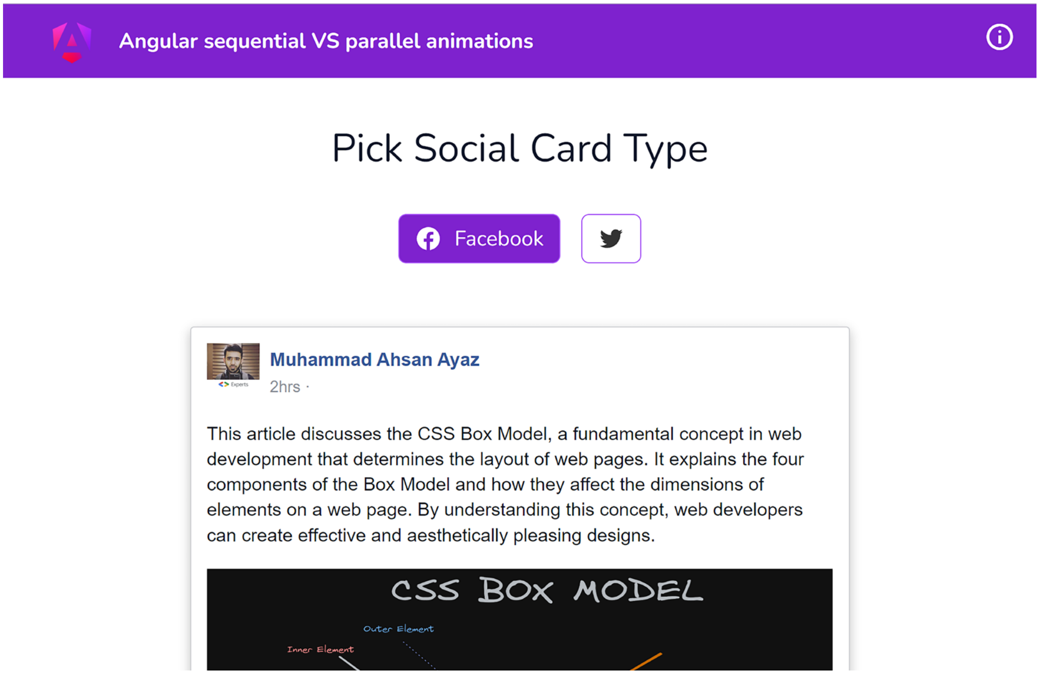
Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-seq-parallel-animations` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-seq-parallel-animations
```

This should open the app in a new browser tab, and you should see the following:



Muhammad Ahsan Ayaz
2hrs ·

This article discusses the CSS Box Model, a fundamental concept in web development that determines the layout of web pages. It explains the four components of the Box Model and how they affect the dimensions of elements on a web page. By understanding this concept, web developers can create effective and aesthetically pleasing designs.

Figure 4.6: ng-seq-parallel-animations app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it…

We have an app that displays the two social cards we used in the previous recipes. One for Facebook and one for Twitter.

To run the animations sequentially and in parallel for both cards, we will use the `query` function to configure the animations sequentially. And then we will use the `group` function to run them in parallel. Let's get started:

1. First, let's provide Angular animations using the `provideAnimations` function from Angular in the `src/app/app.config.ts` file as follows:

```
...
import { provideAnimations } from '@angular/platform-browser/animations';
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(appRoutes,
      withEnabledBlockingInitialNavigation()),
    provideAnimations()
  ],
};
```

2. We will create a simple wrapper transition to handle the cards entering and leaving the DOM. Afterward, we'll handle how they can be triggered together when the current card leaves the view. Create a new file named `animations.ts` in the `app` folder. Add the following code to it:

```
import { trigger, style, transition, animate, query, group, keyframes }
from '@angular/animations';
const duration = '1.5s';
export const cardAnimation = trigger('cardAnimation', [
  transition('* <=> *', [
    // more code here later
  ]),
]);
```

3. Now let's add a query for when the card leaves the view. Inside the `transition` array, add the following `query` as follows:

```
export const cardAnimation = trigger('cardAnimation', [
  transition('* <=> *', [
     query( ':leave', [
        style({ transform: 'translateX(0)', opacity: 1 }),
        animate( `${duration} ease`, style({
            transform: 'translateX(100%)',
          })
        ),
        animate( `${duration} ease`, style({
            opacity: 0,
          })
        ),
        ],
        { optional: true }
      )
    ]),
  ]);
```

4. We will import the animation in the `app.component.ts` file and add it to the `animations` array as follows:

```
...
import { cardAnimation } from './animation';
...
@Component({
  ...
  animations: [cardAnimation],
  ...
})
export class AppComponent {...}
```

5. Now we'll update the `app.component.html` to use the animation on the element with the `card-container` class. Update the file as follows:

```
...
<main>
  …
  <div class="card-container relative h-[600px] w-full
  overflow-hidden py-4"
    [@cardAnimation]="selectedCardType">
    ...
  </div>
</main>
```

   You should be able to see the animation now by clicking the Facebook and Twitter buttons. That is, the card slides from its position to the right of the screen. However, it doesn't look pretty.

6. Let's add another query for when the next card enters the view. We'll first make sure that the card is `invisible` when starting to enter the DOM. Replace the animation in the `animations.ts` file as follows:

```
...
export const cardAnimation = trigger('cardAnimation', [
  transition('* <=> *', [
    query(':enter', [
      style({ opacity: 0 }),
    ]),
    query( ':leave', [
      ...
    ]),
]);
```

7. Now add the second `query` for the card to be entered in the screen. We'll make sure it slides in from the left and becomes visible slowly. Update the `animations.ts` file as follows:

```
...
export const cardAnimation = trigger('cardAnimation', [
  transition('* <=> *', [
    query(':enter', [...]),
    query(':leave', [...]),
    query( ':enter', [
        style({
          transform: 'translateX(-100%)',
          opacity: 0,
        }),
        animate( `${duration} ease`, style({
            transform: 'translateX(0)',
            opacity: 1,
          })
        ),
      ],
      { optional: true }
    ),
  ]),
]);
```

You'll notice that the animations are working now. However, they are really slow. That is, the next card takes a lot of time to appear after the current card has left the screen. That's because they are all running in sequence.

8. We can wrap the second and third queries in a `group` function to run them in parallel. Update the code in the `animations.ts` as follows:

```
import { ..., group } from '@angular/animations';
export const cardAnimation = trigger('cardAnimation', [
  transition('* <=> *', [
    query(':enter', [...]),
    group([
      query( ':leave', [...], { optional: true }),
      query(':enter', [...], { optional: true }),
    ]),
  ]),
]);
```

And boom! You can now see that the animations are running in parallel and are not waiting for the `:leave` transition to be finished before the `:enter` transition is executed.

How it works...

Animations in Angular run in sequence by default. If a transition has more than one step, i.e., `style` and `animate` usages, the animations will run in sequence. The `group` function makes it possible for us to run animations in parallel. For this recipe, we wanted both `:enter` and `:leave` transitions to run at the same time so we grouped them to run in parallel at the end.

See also

- Sequential vs parallel animations in Angular (`https://angular.io/guide/complex-animation-sequences#sequential-vs-parallel-animation`)
- Angular animations `sequence` function (`https://angular.io/api/animations/sequence`)

Route animations in Angular

In this recipe, you'll learn how to implement route animations in Angular. You'll learn how to configure route animations by passing the transition state name to the route as a data property. You'll also learn how to use the `RouterOutlet` API to get the transition name and apply it to the animation to be executed. We'll implement some 3D transitions, so it is going to be fun!

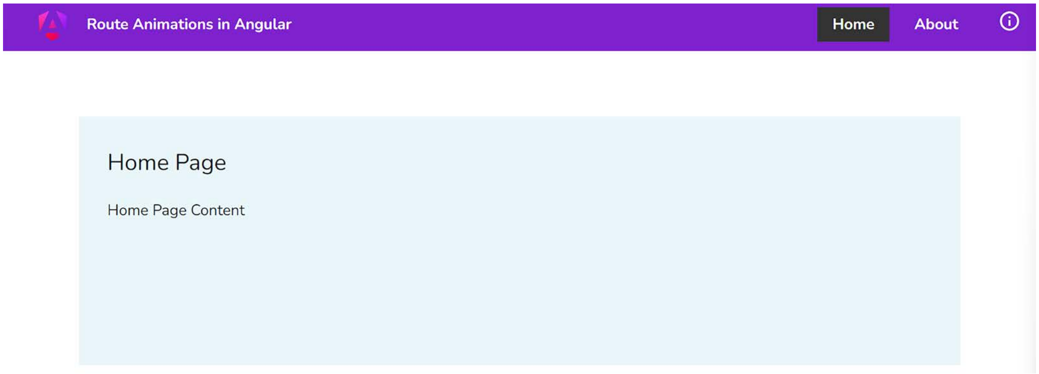
Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-route-animations` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-route-animations
```

This should open the app in a new browser tab, and you should see the following:



Home Page

Home Page Content

Figure 4.7: ng-route-animations app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

We have a simple app with two lazy-loaded routes now. The routes are for the **Home** and the **About** pages, and we will now start configuring the animations for the app:

1. First, let us provide Angular animations using the `provideAnimations` function from Angular in the `src/app/app.config.ts` file as follows:

```
...
import { provideAnimations } from '@angular/platform-browser/animations';
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(appRoutes,
      withEnabledBlockingInitialNavigation()),
    provideAnimations()
  ],
};
```

2. We will now create a new file named `animations.ts` inside the app folder. Let us put the following code in the `animations.ts` file to register a basic trigger to handle animations from a route to every other route:

```
import {trigger,  transition } from '@angular/animations';
export const ROUTE_ANIMATION = trigger('routeAnimation', [
  transition('* <=> *', [
    // states and transitions to be added here
  ])
]);
```

3. We'll now register some queries and basic states for the animations. Let's add the following items in the `transition` function's array as follows:

```
import { trigger, transition, style, query } from '@angular/animations';
const optional = { optional: true };

export const ROUTE_ANIMATION = trigger('routeAnimation', [
  transition('* <=> *', [
    style({ position: 'relative',
      perspective: '1000px' }),
    query(
      ':enter, :leave',
      [style({ position: 'absolute', width: '100%' })],
      optional
      ),
    ]),
  ]);
```

Alright! We have the `routeAnimation` trigger registered now for transitioning from a route to every other route. Now, let's provide those transition states in the routes.

4. We can provide the states for the transitions using a unique identifier for each route. There are many ways to do this, but the easiest way is to provide them using the `data` attribute in the route configuration as follows in the `app.routes.ts` file:

```
export const appRoutes: Route[] = [
  ...,
  {
    path: 'home',
    data: { transitionState: 'HomePage' },
    loadComponent: () =>
      import('./home/home.component').then(
        (m) => m.HomeComponent),
  },
  {
    path: 'about',
    data: { transitionState: 'AboutPage' },
    loadComponent: () =>
      import('./about/about.component').then(
        (m) => m.AboutComponent),
  },
];
```

5. Now, we need to provide this `transitionState` property from the current route to the route animation trigger in the `app.component.html` file. For this, create an `@ViewChild` property in the `app.component.ts` file. This `ViewChild` is going to target the `<router-outlet>` element in the `app.component.html` template. This is so we can get the current route's `data` and the `transitionState` value provided. The code in the `app.component.ts` file should look as follows:

```
import { CommonModule } from '@angular/common';
import { Component, ViewChild } from '@angular/core';
import { RouterModule, RouterOutlet } from '@angular/router';
export class AppComponent {
  @ViewChild(RouterOutlet) routerOutlet!: RouterOutlet;
}
```

6. We'll also import `ROUTE_ANIMATION` from the `animations.ts` file into `app.component.ts` as follows:

   ```
   ...
   import { ROUTE_ANIMATION } from './animations';
   @Component({
     selector: "app-root",
     templateUrl: "./app.component.html",
     styleUrls: ["./app.component.scss"],
     animations: [
       ROUTE_ANIMATION
     ]
   })
   ```

   We'll now create a method named `getRouteAnimationState`, which will get the current route's data and the `transitionState` value and return it. This function will later be used in `app.component.html`. Modify your code in `app.component.ts` as follows:

   ```
   ...
   @Component({
   ...
   })
   export class AppComponent {
     @ViewChild(RouterOutlet) routerOutlet!: RouterOutlet;
     getRouteAnimationState() {
       return (
         this.routerOutlet &&
         this.routerOutlet.activatedRouteData &&
         this.routerOutlet.activatedRouteData['transitionState']
       );
     }
   }
   ```

7. Finally, let's use the `getRouteAnimationState` method with the `@routeAnimation` trigger in `app.component.html` so we can see the animation in play:

   ```
   ...
   <div class="content" role="main">
     <div class="router-container"
       [@routeAnimation]="getRouteAnimationState()">
       <router-outlet></router-outlet>
     </div>
   </div>
   ```

8. Now that we have everything set up, let's finalize the animations. We'll add a query for when a route leaves the view. Update the `animations.ts` file as follows:

```
import { trigger, style, transition, query, animate, keyframes } from '@
angular/animations';
...
export const ROUTE_ANIMATION = trigger('routeAnimation', [
  transition('* <=> *', [
    style({ position: 'relative', perspective: '1000px' }),
    query( ':enter, :leave', [...] ,optional),
    query( ':leave', [
        animate( '1s ease-in', keyframes([
            style({ opacity: 1, offset: 0,
              transform: 'rotateY(0) translateX(0)
              translateZ(0)',
            }),
            style({ offset: 0.25, transform:
              'rotateY(45deg) translateX(25%)
              translateZ(100px) translateY(5%)',
            }),
            style({ offset: 0.5, transform:
              'rotateY(90deg) translateX(75%)
              translateZ(400px) translateY(10%)',
            }),
            style({ offset: 0.75, transform:
              'rotateY(135deg) translateX(75%)
              translateZ(800px) translateY(15%)',
            }),
            style({ opacity: 0, offset: 1, transform:
              'rotateY(180deg) translateX(0)
              translateZ(1200px) translateY(25%)',
            }),
          ])
        ),
      ],
      optional
    ),
  ]),
]);
```

If you navigate between routes, you'll notice the leaving route going out with animation behind the entering route. Let's add the animation for the entering route as well.

9. We'll add the animation for the route entering the view. Update the `animations.ts` as follows:

```
...
export const ROUTE_ANIMATION = trigger('routeAnimation', [
  transition('* <=> *', [
    style({ position: 'relative', perspective: '1000px' }),
    query(':enter, :leave', ...),
    query(':leave', ...),
    query( ':enter', [
        animate( '1s ease-out', keyframes([
            style({ opacity: 0, offset: 0, transform:
              'rotateY(180deg) translateX(25%) translateZ(1200px)',
            }),
            style({ offset: 0.25, transform:
              'rotateY(225deg) translateX(-25%) translateZ(1200px)',
            }),
            style({ offset: 0.5, transform:
              'rotateY(270deg) translateX(-50%) translateZ(400px)',
            }),
            style({ offset: 0.75, transform:
              'rotateY(315deg) translateX(-50%) translateZ(25px)',
            }),
            style({ opacity: 1, offset: 1, transform:
              'rotateY(360deg) translateX(0) translateZ(0)',
            }),
          ])
        ),
      ],
      optional
    ),
  ]),
]);
```

If you look at the animation route while navigating, you'll notice that the entering route appears immediately, then we see the leaving route's animation, and after that, we see the entering route's animation. Let's group the entering and leaving animations together to run them in parallel.

10. Update the `animations.ts` as follows to run the entering and leaving routes' animations in parallel:

    ```
    import {..., group } from '@angular/animations';
    export const ROUTE_ANIMATION = trigger('routeAnimation', [
      transition('* <=> *', [
        style({ position: 'relative', perspective: '1000px' }),
        query( ':enter, :leave', ...),
        group([
          query( ':leave', [...], optional ),
          query( ':enter', [...], optional),
        ]),
      ]),
    ]);
    ```

Voilà! Refresh the app and see the magic. You should now see the 3D animation for both the entering and leaving routes as you navigate from the **Home** page to the **About** page and vice versa. The sky's the limit when it comes to what you can do with keyframes and animations in Angular.

How it works...

In the `animations.ts` file, we first define our animation trigger named `routeAnimation`. Then we make sure that by default, the HTML element to which the trigger is assigned has `position: 'relative'` set as a style:

```
transition('* <=> *', [
    style({
      position: 'relative'
    }),
    ...
])
```

Then we apply the styled `position: 'absolute'` to the children, as mentioned, using `:enter` and `:leave` as follows:

```
    query(':enter, :leave', [
      style({
        position: 'absolute',
        width: '100%'
      })
    ], {optional: true}),
```

This makes sure that these elements, that is, the routes to be loaded, have the `position: 'absolute'` style and a full width using `width: '100%'` so they can appear on top of each other. You can always play around with it by commenting on either of the styles to see what happens (at your own risk, though!).

Then we defined our route transitions as a combination of two animations, the first for `query :leave` and the second for `query :enter`. For the route leaving the view, we set the `opacity` to `0` via the animation, and for the route entering the view, we set the `opacity` to `1` via the animation as well. Note that animations via Angular animations run in sequence:

```
    query(':leave', [
      ...
    ], {optional: true}),
    query(':enter', [
      ...
    ], {optional: true}),
```

You'll notice that in our code, we're using the `keyframes` function for animations. For the leaving route, The "`keyframes` function" starts with `opacity 1`, and without any transformations initially. And then it ends with `opacity 0` but with the transform set to `'rotateY(180deg) translateX(0) translateZ(1200px) translateY(25%)'`. This is the opposite of the entering route.

Finally, we use the `group` function to wrap both the leaving and entering animations together so they can run in parallel instead of in sequence. This makes it so that the entering route comes in as the leaving route goes away.

See also

- Animations in Angular (`https://angular.io/guide/animations`)
- Angular route transition animations (`https://angular.io/guide/route-animations`)

Disabling Angular animations conditionally

In this recipe, you'll learn how to disable animations in Angular conditionally. This is useful for a variety of cases, including disabling animations on a particular device, for example.

> Pro tip: Use `ngx-device-detector` to identify if your Angular app is running on a mobile, tablet, etc. (A little secret that is not so-secret anymore...I built it!)

Shameless plug aside, in this recipe, we'll disable animations for employees in the application, considering we're rolling out animations only for admins at the moment.

Getting ready

The app that we are going to work with resides in `start/apps/chapter04/ng-disable-animations` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-disable-animations
```

This should open the app in a new browser tab. Log in as an admin, add a few bucket items, and you should see the following:

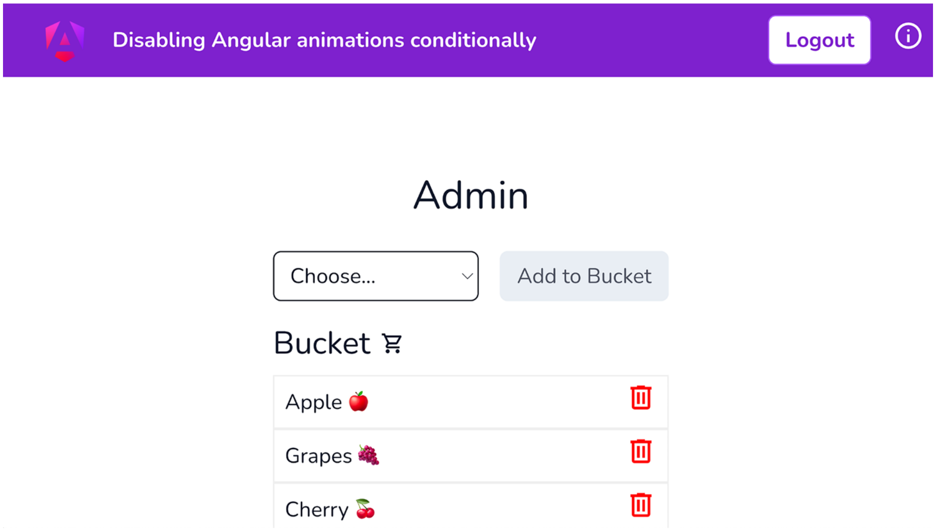


Figure 4.8: ng-disable-animations app running on http://localhost:4200

Now that we have the app running, we will move on to the steps for the recipe.

How to do it...

We have an app that already has some Angular animations configured. You will notice that both the admin and employee pages have animations enabled. We will disable the animations for the employee page using a `config`. Let's continue with the steps as follows:

1. First, we'll add a new property named `disableAnimations` to our `IEmployeeConfig` interface and the `EMPLOYEE_CONFIG` variable in the `src/app/app.config.ts` file as follows:

```
...
export interface IEmployeeConfig {
  canDeleteItems: boolean;
  disableAnimations: boolean;
}
...
export const employeeConfig: IEmployeeConfig = {
  canDeleteItems: true,
  disableAnimations: false
};
...
```

   If you save the file, TypeScript will start throwing errors in the console as we need to add the same `disableAnimations` property in the `employee.config.ts` file as well.

2. Update the `src/app/employee/employee.config.ts` file as follows:

```
import { IEmployeeConfig } from '../app.config';

export const employeeConfig: IEmployeeConfig = {
  canDeleteItems: false,
  disableAnimations: true
};
```

3. Finally, add a `HostBinding` in the `bucket.component.ts` to disable the animations in the bucket component based on the config. Update the `bucket/bucket.component.ts` file as follows:

```
import { CommonModule } from '@angular/common';
import { Component, inject, OnInit, HostBinding } from '@angular/core';
...
@Component({...})
export class BucketComponent implements OnInit {
  ...
  fruits: string[] = Object.values(Fruit);
  @HostBinding('@.disabled')
  animationsDisabled = this.appConfig.disableAnimations;

  ngOnInit(): void {...}
        ...
}
```

Great! If you now refresh the app and view the **Admin** page, you'll see the animations working. And if you go to the **Employee** page, you'll see that the animations are disabled there. Magic! Check out the next section to understand how the recipe works.

How it works...

Angular provides a way to disable animations using the `[@.disabled]` binding. You can put it anywhere in the template with an expression that evaluates a truth value. And in that case, all the child animations that are applied in its nested HTML tree will be disabled. We had an application-level config that is overridden in the employee component via the `EmployeeConfig` object. So first we created a `disableAnimations` property in the `IAppConfig` interface. This interface is used by both the `AppConfig` variable in the `app-config.ts` file and the `employeeConfig` variable in the `employee.config.ts` file. As you can see, we set the value of `disabledAnimations` to `false` for the configuration defined in the `app.config.ts` and `true` for the configuration defined in `employee.config.ts`. And then we used the `@HostBinding()` decorator in the `BucketComponent` class by assigning its value to the provided config's `disabledAnimations` property. Since the `AdminComponent` class gets the configuration defined in the `app.config.ts` against the `EMPLOYEE_CONFIG` token, and the `EmployeeComponent` class gets the configuration defined in the `employee.config.ts` against the `EMPLOYEE_CONFIG` token, the animations are enabled and disabled, respectively, for these components.

See also

- Animations in Angular: `https://angular.io/guide/animations`
- Angular Animations Explained with Examples: `https://www.freecodecamp.org/news/angular-animations-explained-with-examples/`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

5

Angular and RxJS – Awesomeness Combined

Angular and **RxJS** create a killer combination of awesomeness. By combining these, you can handle your data reactively, work with streams, and implement complex business logic in your Angular apps. That's exactly what you're going to learn about in this chapter.

Here are the recipes we're going to cover in this chapter:

- Sequential and parallel HTTP calls in Angular with RxJS
- Listening to multiple observable streams
- Unsubscribing streams to avoid memory leaks
- Using Angular's `async` pipe to unsubscribe streams automagically
- Using the `map` operator to transform data
- Using the `switchMap` and `debounceTime` operators with autocompletes for better performance
- Creating a custom RxJS operator
- Retrying failed HTTP calls with RxJS

Technical requirements

For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter05`.

Sequential and parallel http calls in Angular with RxJS

In this recipe, you'll learn how to use different RxJS operators to make sequential and parallel HTTP calls in Angular apps. We'll work with the famous Star Wars API (**swapi**) to get some data to display on the UI.

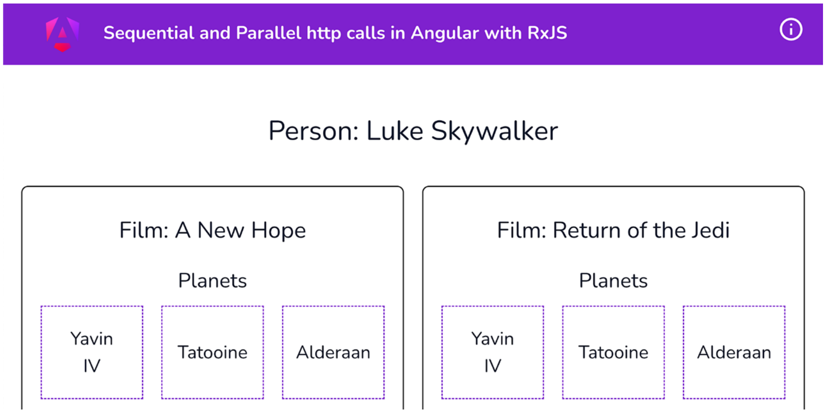
Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-seq-parallel-http` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve rx-seq-parallel-http
```

This should open the app in a new browser tab, and you should see the following:



Sequential and Parallel http calls in Angular with RxJS
Person: Luke Skywalker
Film: A New Hope
Planets
Yavin IV
Tatooine
Alderaan
Film: Return of the Jedi
Planets
Yavin IV
Tatooine
Alderaan

Figure 5.1: The rx-seq-parallel-http app running on http://localhost:4200

Now that we have the app running, we will move on to the steps for the recipe.

How to do it...

We have an Angular app that uses the Star Wars API (swapi) to fetch a person from Star Wars, and the films the person has been in. All of this happens using a lot of HTTP calls, and our code is totally garbage so far. This is because we first show the loader, but we hide it before we've retrieved all the data. Also, if you keep refreshing the page, you'll see that the sequence of the films can change each time. Hence, we see the UI jumping a lot. Our desired approach is to fetch the person, then fetch all the films, and then hide the loader. We will implement this approach using RxJS. Let's begin:

1. First, we'll avoid using the `setTimeout` function to rely on the person data being available in `1500ms`. We would rather move this inside the `subscribe` block and will also handle errors appropriately. Update the `fetchData` method inside the `app.component.ts` as follows:

```
fetchData() {
    this.loadingData = true;
    this.swapi.fetchPerson('1').subscribe({
      next: (person) => {
        this.person = person;
        this.person.filmObjects = [];
```

```
          this.person.films.forEach((filmUrl) => {
            this.swapi.fetchFilm(filmUrl).subscribe({
              next: (film) => {
                this.person.filmObjects.push(film);
                this.loadingData = false;
              },
              error: (err) => {
                console.error('Error while fetching film',
                  err);
              },
            });
          });
      },
      error: (err) => {
        console.error('Error while fetching person', err);
      }
    });
  }
```

This has a potential problem. And that is, as soon as the first film is retrieved, the loader is hidden because we set `this.loadingData` to `false`.

2. Now, we will add the `mergeMap` operator using the `pipe` method to be able to chain calls later. For now, we will just move the code for adding the `filmObjects` array in the `this.person` object inside the `mergeMap` callback. Update the `fetchData` method as follows now:

```
...
import { mergeMap, of } from 'rxjs';
...
  fetchData() {
    this.loadingData = true;
    this.swapi
      .fetchPerson('1')
      .pipe(
        mergeMap((person) => {
          const personObj = {
            ...person,
            filmObjects: [],
          };
          return of(personObj);
        })
      )
      .subscribe({
        next: (person) => {
          this.person = person;
          this.person.films.forEach((filmUrl) => {
            this.swapi.fetchFilm(filmUrl).subscribe({
              next: (film) => {
                this.person.filmObjects.push(film);
                this.loadingData = false;
              },
              error: (err) => {
                console.error('Error while fetching film', err);
              },
            });
          });
        },
        error: (err) => {
          console.error('Error while fetching person', err);
        },
      });
  }
```

Notice that the UI becomes *a bit better*. The loader still hides as soon as one of the films is retrieved from the server. However, we want to hide the loader when all of them have been retrieved. Also, the sequence of the films is still unpredictable.

3. Now, we'll use the `forkJoin` function to make the API calls for the films in parallel and to wait for the combined response. We're doing it instead of the `of` operator, which just passes the film URLs from the `mergeMap` function. Update the `fetchData` method as follows and update the imports at the top:

```
import { forkJoin, mergeMap, of (//<-- remove of) } from 'rxjs';
...
  fetchData() {
    this.loadingData = true;
    this.swapi
      .fetchPerson('1')
      .pipe(
        mergeMap((person) => {
          const personObj = {
            ...person,
            filmObjects: [],
          };
          this.person = personObj;
          return forkJoin(
            this.person.films.map((filmUrl) =>
              this.swapi.fetchFilm(filmUrl))
          );
        }),
        catchError((err) => {
          console.error('Error while fetching films', err);
          alert('Could not get films. Please try again.');
          return of([]);
        })
      )
      .subscribe({
        next: (films) => {
          this.person.filmObjects = films;
          this.loadingData = false;
        },
        error: (err) => {
          console.error('Error while fetching person', err);
        },
      });
  }
```

Woohoo! Now if you refresh the app, you'll notice two things. First, the loader always stops only when all the data has been fetched. And second, the sequence of the films is always the same (and correct).

Now that you've finished the recipe, let's move on to the next section to understand how it all works.

How it works...

The `mergeMap` operator allows us to chain observables by returning an **observable** from its callback. You can think of it like how we chain `Promise.then`, but this is for observables. A popular alternative is the `switchMap` operator, which works like the `mergeMap` operator but also cancels the previous call/execution if called twice or more before the first call/execution completes. We first removed the `setTimeout` function (which is mostly nonsensical to put into the code for these cases because the result is not always predictable with respect to time) and moved the logic of fetching films inside the `subscribe` block of fetching the person. We also used the `of` operator to return the `personObject` object from the `mergeMap` function's callback. The `mergeMap` function is used to chain observables together and, in our context, it can be chained to wait for one HTTP call to finish so we can execute the other ones. In *step 3*, we intend to execute multiple HTTP calls together in parallel for all the films for the person. We do this using the `forkJoin` operator, which takes an array of observables. In this case, those observables are HTTP calls for each film. `forkJoin` also makes it possible to wait for all the parallel calls to finish and then trigger the `subscribe` block's callback. One more thing `forkJoin` does is it provides us with the responses in the form of an array having the responses in the same sequences as the observables. This makes the responses predictable, and we always show the same data on the UI.

See also

- *Catch the Dot Game*—RxJS documentation (`https://www.learnrxjs.io/learn-rxjs/recipes/catch-the-dot-game`)
- RxJS `mergeMap` operator documentation (`https://www.learnrxjs.io/learn-rxjs/operators/transformation/mergemap`)
- RxJS `merge` operator documentation (`https://www.learnrxjs.io/learn-rxjs/operators/combination/forkjoin`)

Listening to multiple observable streams

In this recipe, we'll work with the `combineLatest` operator to listen to multiple observable streams at once. Using this operator results in having an array as an output, combining all the streams. This approach is appropriate for when you want the latest output from all the streams, combined in a single subscription.

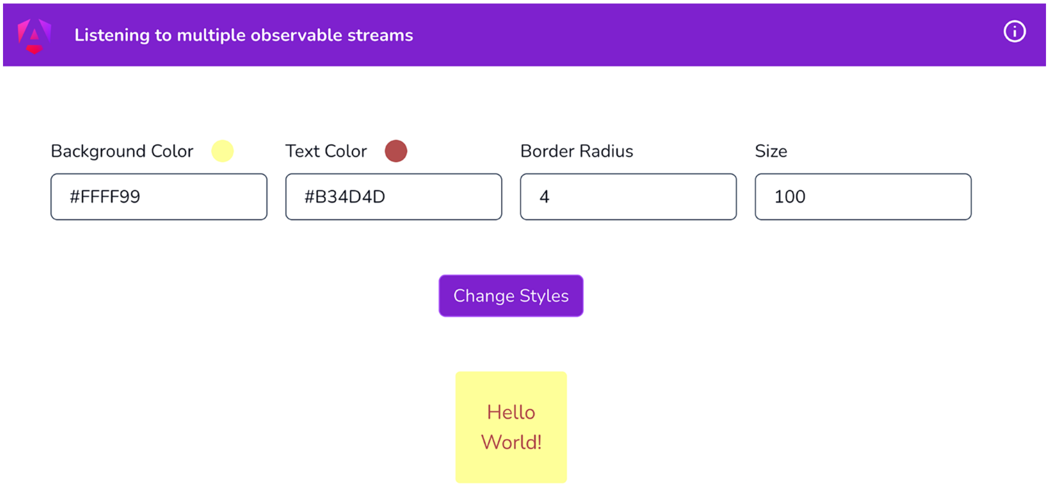
Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-multiple-streams` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve rx-multiple-streams
```

This should open the app in a new browser tab, and you should see the following:



Background Color: #FFFF99

Text Color: #B34D4D

Border Radius: 4

Size: 100

Figure 5.2: The rx-multiple-streams app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

For this recipe, we have an app that displays a box. The box has a size (width and height), a border radius, a background color, and a color for its text. It also has four inputs using the **Reactive Forms** API to modify all the mentioned factors. Right now, we have to apply the changes manually with the click of a button even if the inputs change. What if we could subscribe to the changes to the inputs and update the box right away without having the user click the button? That's what we're going to do here:

1. We'll begin by creating a method named `listenToInputChanges`. We'll create an array of controls we want to work with. Update the code of `home.component.ts` as follows:

```
...
export class HomeComponent implements OnInit {
  ...
  ngOnInit() {
    this.applyChanges();
  }

  listenToInputChanges() {
    const controls: AbstractControl[] = [
      this.boxForm.controls.size,
      this.boxForm.controls.borderRadius,
      this.boxForm.controls.textColor,
      this.boxForm.controls.backgroundColor,
    ];
  }

  ...
}
```

2. Now we'll loop over the controls to give them the initial value so that when the observable stream is subscribed, they have a value to work with. Update the `listenToInputChanges` method further as follows:

```
import { startWith } from 'rxjs';
...
export class HomeComponent implements OnInit {
...
listenToInputChanges() {
    const controls: AbstractControl[] = [...];
    controls.map((control) =>
      control.valueChanges.pipe(startWith(control.value))
    );
  }
}
```

3. Now we'll replace the `boxStyles` property with an `Observable` named `boxStyles$`. Then we'll wrap the `valueChanges` streams of each form control inside a `combineLatest` operator to join them. Finally, we'll assign the result of this joined stream to the `boxStyles$` observable. Update the `home.component.ts` file as follows:

```
...

import { Observable, combineLatest, startWith} from 'rxjs';
...
export class HomeComponent implements OnInit, OnDestroy {
  ...
  boxStyles$!: Observable<BoxStyles>;
        ...
  listenToInputChanges() {
    this.boxStyles$ = combineLatest(
      controls.map((control) =>
        control.valueChanges.pipe(startWith(control.value))
      );
    );
  }
  ...
}
```

4. Now we will use a `map` operator with `pipe` on the combined stream to map it to the `BoxStyle` type values. Update the `listenToInputChanges` method in the `home/home.component.ts` file, as follows:

```
import { combineLatest, map, Observable, startWith } from 'rxjs';
export class HomeComponent implements OnInit {
  listenToInputChanges() {
    const controls: AbstractControl[] = [...];
    this.boxStyles$ = combineLatest(...).pipe(
      map(([size, borderRadius, textColor,
        backgroundColor]) => {
        return { width: `${size}px`, height: `${size}px`,
          backgroundColor: backgroundColor,
          color: textColor,
          borderRadius: `${borderRadius}px`,
        };
      })
    );
  }
}
```

5. We need to remove the `setBoxStyles` and `applyChanges` methods and the usages of the `applyChanges` method from the `home.component.ts` file. Update the file, as follows:

```
export class HomeComponent implements OnInit {
  ...
  ngOnInit() {
    this.listenToInputChanges(); //← Add this call
    ...
    this.applyChanges(); //← Remove this call
  }
  ...
  setBoxStyles(...) {...}  //← Remove this method
  applyChanges() {...} //← Remove this method
  ...
}
```

6. We also need to remove the usage of the `applyChanges` method from the template as well. Remove the `(ngSubmit)` handler from the `<form>` element in the `home.component.html` file so that it looks like this:

```
<div class="home" [formGroup]="boxForm"
  (ngSubmit)="applyChanges()" <!--← Remove this-->
  ...
</div>
```

7. We also need to get rid of the `submit-btn-container` element from the `home.component.html` template as we don't need it anymore. Delete the following chunk from the file:

```
<div class="row submit-btn-container" <!--← Remove this element -->
  <button class="btn btn-primary" type="submit"
    (click)="applyChanges()">Change Styles</button>
</div>
```

8. Now that we can work with the `boxStyles$` Observable, let's use it in the template, i.e., the `home.component.html` file, instead of the `boxStyles` property:

```
  ...
  <div class="row" *ngIf="boxStyles$ | async as
    boxStyles">
    <div class="box" [ngStyle]="boxStyles">
      <div class="box__text">
        Hello World!
      </div>
    </div>
  </div>
  ...
```

And voilà! If you refresh the app, you should be able to see the box appearing with the default styles. And if you change any of the options, you'll see the changes reflected as well.

Congratulations on finishing the recipe. You're now the master of handling multiple streams using the `combineLatest` operator. See the next section to understand how it works.

How it works…

The beauty of Reactive Forms is that they provide much more flexibility than the regular `ngModel` binding or even template-driven forms. And for each form control, we can subscribe to its `valueChanges` observable, which receives a new value whenever the input is changed. So, instead of relying on the **Submit** button's click, we subscribe directly to the `valueChanges` property of each **form control**. In a regular scenario, that would result in four different streams for four inputs, which means we would have four subscriptions that we need to take care of and make sure we unsubscribe them. This is where the `combineLatest` operator comes into play. We used the `combineLatest` operator to combine those four streams into one, which means we needed to unsubscribe only one stream on component destruction. But hey! Remember that we don't need to do this if we use the `async` pipe? That's exactly what we did. We removed the subscription from the `home.component.ts` file and used the `pipe` method with the `map` operator. The `map` operator transformed the data according to our needs and then returned the transformed data to be set to the `boxStyles$` observable. Finally, we used the `async` pipe in our template to subscribe to the `boxStyles$` observable and assigned its value as `[ngStyle]` to our box element. Since `valueChanges` is a `Subject` instead of a `ReplaySubject`, we also piped a `startWith` with `valueChanges` to provide an initial value. If we don't use `startWith`, the box won't show unless all the inputs have a value changed manually at least once. Try it!

See also

- `combineLatest` operator documentation (`https://www.learnrxjs.io/learn-rxjs/operators/combination/combinelatest`)
- Visual representation of the `combineLatest` operator (`https://rxjs-dev.firebaseapp.com/api/index/function/combineLatest`)

Unsubscribing streams to avoid memory leaks

Streams are fun to work with and they're awesome. You'll know more about RxJS and streams when you've finished this chapter. One reality is facing unseen problems that occur when streams are used without caution. One of the biggest mistakes to make with streams is to not unsubscribe them when we no longer need them, and in this recipe, you'll learn how to unsubscribe streams to avoid memory leaks in your Angular apps.

Getting ready

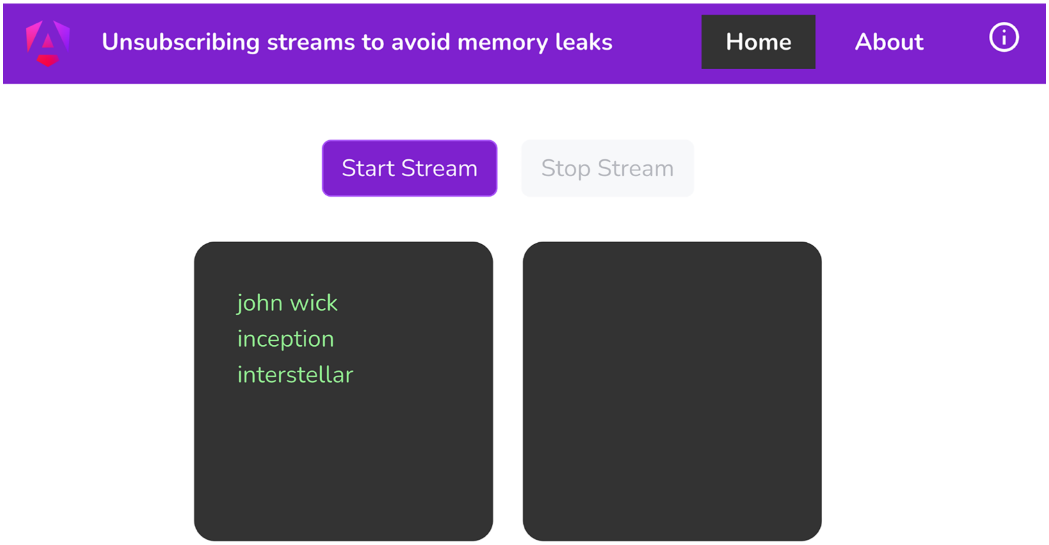
The app that we are going to work with resides in `start/apps/chapter05/rx-unsubscribing-streams` inside the cloned repository:

1. Open the code repository in your code editor.

2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve rx-unsubscribing-streams
   ```

   This should open the app in a new browser tab, and you should see the following:



john wick
inception
interstellar

Figure 5.3: The rxjs-unsubscribing-streams app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

We currently have an app with two routes—that is, **Home** and **About**. This is to show you that unhandled subscriptions can cause memory leaks in an app. The default route is **Home**, and in the `HomeComponent` class, we handle a single stream that outputs data using the `interval` operator function:

1. Tap the **Start Stream** button, and you should see the stream emitting values.
2. Then, navigate to the **About** page by tapping the **About** button from the header (top right), and then come back to the **Home** page.

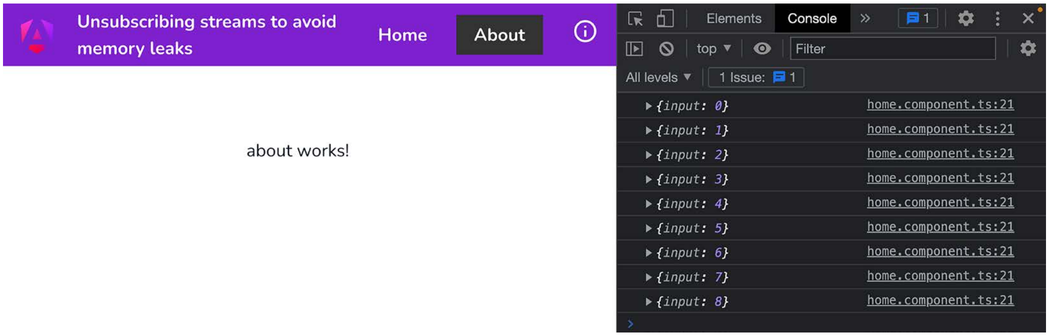
   Do you see anything weird? No? Everything looks fine, right? Well, not exactly.

3. To see whether we have an unhandled subscription, let's put `console.log` inside the `startStream` method in the `home.component.ts` file—specifically, inside the `subscribe` function's block, as follows:

   ```
   ...
   export class HomeComponent implements OnInit {
     ...
     startStream() {
       const streamSource = interval(1500);
   ```

```
    this.subscription = streamSource.subscribe((input) => {
      this.outputStreamData.push(input);
      console.log({ input });
    });
  }
  stopStream() {...}
}
```

If you now perform the same steps as mentioned in *Step 1*, you'll see the following output on the console, as shown in *Figure 5.4*:



about works!

```
{input: 0}                    home.component.ts:21
{input: 1}                    home.component.ts:21
{input: 2}                    home.component.ts:21
{input: 3}                    home.component.ts:21
{input: 4}                    home.component.ts:21
{input: 5}                    home.component.ts:21
{input: 6}                    home.component.ts:21
{input: 7}                    home.component.ts:21
{input: 8}                    home.component.ts:21
```

Figure 5.4: interval emitting values on the About page

Want to have some more fun? Try performing *Step 1* a couple of times without refreshing the page even once. What you'll see will be *chaos!*

4. So, to solve the issue, we'll use the simplest approach—that is, unsubscribing the stream when the user navigates away from the route. Let's implement the `ngOnDestroy` lifecycle method for that, as follows:

```
import { Component, OnDestroy } from '@angular/core';
...
@Component({...})
export class HomeComponent implements OnDestroy {
  ...
  startStream() {...}
  ngOnDestroy() {
    this.stopStream();
  }
  stopStream() {...}
}
```

Great! If you follow the instructions from *Step 1* again, you'll see that there's no further log on the console once you navigate away from the **Home** page, and our app doesn't have an unhandled stream causing memory leaks now. Read the next section to understand how it works.

How it works...

When we create an `Observable/stream` and we subscribe to it, RxJS automagically adds our provided `subscribe` function block as a handler to the `Observable`. So, whenever there's a value emitted from the `Observable`, our method is supposed to be called. The fun part is that Angular doesn't automatically destroy that subscription/handler when the component unmounts or when you have navigated away from the route. That's because the core of observables is **RxJS**, not Angular; therefore, it isn't Angular's responsibility to handle it.

Angular provides certain lifecycle methods, and we used the `OnDestroy (ngOnDestroy)` method. So, we used the `ngOnDestroy` method to call the `stopStream` method so that the subscription is destroyed as soon as the user navigates away from the page. This is possible because, when we navigate away from a route, Angular destroys that route and thus we can execute our `stopStream` method.

There's more...

In a complex Angular app, there will be cases where you'd have more than one subscription in a component, and when the component is destroyed, you'd want to clean all those subscriptions at once. Similarly, you might want to unsubscribe based on certain events/conditions rather than the `OnDestroy` lifecycle. Here is an example where you have multiple subscriptions in hand, and you want to clean up all of them together when the component is destroyed:

```
startStream() {
    const streamSource = interval(1500);
    const secondStreamSource = interval(3000);
    const fastestStreamSource = interval(500);
    streamSource.subscribe((input) => {
      this.outputStreamData.push(input);
      console.log('first stream output', input);
    });
    secondStreamSource.subscribe(input => {
      this.outputStreamData.push(input);
      console.log('second stream output', input)
    });

    fastestStreamSource.subscribe(input => {
      this.outputStreamData.push(input);
      console.log('fastest stream output', input)
    });
  }
  stopStream() {
    // remove code from here
  }
```

Notice that we're not saving the **subscription** from `streamSource` to `this.subscription` anymore, and we have also removed the code from the `stopStream` method. The reason for this is that we don't have individual properties/variables for each subscription. Instead, we'll have a single variable to work with. Let's look at the following recipe steps to get things rolling:

1. First, we'll create a property in the `HomeComponent` class named `isStreamActive`:

```
import { Component, OnDestroy } from '@angular/core';
  ...
export class HomeComponent implements OnDestroy {
  isStreamActive = true;
  ...
}
```

2. Now, we'll import the `takeWhile` operator from `rxjs/operators`, as follows:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
...
import { interval, Subscription, takeWhile } from 'rxjs';
```

3. We'll now use the `takeWhile` operator with each of our streams to make them work only when the `isStreamActive` property is set to `true`. Since `takeWhile` takes a `predicate` method, it should look like this:

```
startStream() {
    ...
    streamSource
      .pipe(takeWhile(() => this.isStreamActive))
      .subscribe(input => {...});
    secondStreamSource
      .pipe(takeWhile(() => this.isStreamActive))
      .subscribe(input => {...});
    fastestStreamSource
      .pipe(takeWhile(() => this.isStreamActive))
      .subscribe(input => {...});
  }
```

If you click the **Start Stream** button right now on the **Home** page, you still won't see any output or logs because the `isStreamActive` property is still **undefined**.

4. To make the streams work, we'll set the `isStreamActive` property to `true` in the `startStream` method. The code should look like this:

```
  ngOnDestroy() {
    this.stopStream();
  }
```

```
startStream() {
isStreamActive = true;
  const streamSource = interval(1500);
  const secondStreamSource = interval(3000);
  const fastestStreamSource = interval(500);
  ...
}
```

After this step, if you now try to start the stream and navigate away from the page, you'll still see the same issue with the streams—that is, they've not been unsubscribed.

5. To unsubscribe all streams at once, we'll set the value of `isStreamActive` to `false` in the `stopStream` method, as follows:

```
stopStream() {
  this.isStreamActive = false;
}
```

6. Finally, update the template to handle which button is disabled based on the `isStreamActive` property instead of `subscription`. Update the `home.component.html` file as follows:

```
<div class="home">
    <div class="buttons-container">
      <button [disabled]="isStreamActive" class="btn btn-
        primary" (click)="startStream()">Start
        Stream</button>
      <button [disabled]="!isStreamActive" class="btn
        btn-dark" (click)="stopStream()">Stop
        Stream</button>
    </div>
    ...
</div>
```

And boom! Now, if you navigate away from the route while the streams are emitting values, the streams will stop immediately as soon as you navigate away from the **Home** route. Voilà!

See also

- Read about RxJS subscription (`https://www.learnrxjs.io/learn-rxjs/concepts/rxjs-primer#subscription`)
- `takeWhile` operator docs (`https://www.learnrxjs.io/learn-rxjs/operators/filtering/takewhile`)

Using Angular's async pipe to unsubscribe streams automatically

As you learned in the previous recipe, it is crucial to unsubscribe the streams you subscribe to. What if we had an even simpler way to unsubscribe them when the component gets destroyed—that is, letting Angular take care of it somehow? In this recipe, you'll learn how to use Angular's `async` pipe with an observable to directly bind the data in the stream to the Angular template instead of having to subscribe in the `*.component.ts` file.

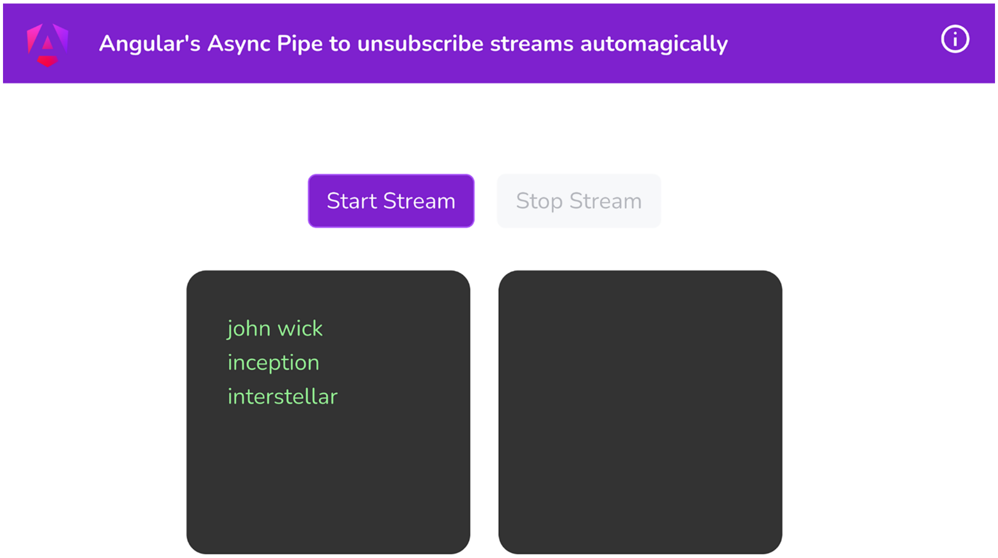
Getting ready

The app that we are going to work with resides in `start/apps/chapter05/ng-async-pipe` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-async-pipe
```

This should open the app in a new browser tab, and you should see the following:



john wick
inception
interstellar

Figure 5.5: The ng-async-pipe app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it…

The app we have right now has three streams/observables observing values at different intervals. We're relying on the `isStreamActive` property to keep the subscription alive or make it stop when the property is set to `false`. We'll remove the usage of `takeWhile` and somehow make everything work similarly to what we have right now.

1. First, add a property of type `Observable` in the `HomeComponent` class named `streamOutput$`. Update the code in the `home.component.ts` file as follows:

```
...
import { interval, Observable, takeWhile } from 'rxjs';
...
export class HomeComponent implements OnDestroy {
  ...
  isStreamActive!: boolean;
  streamsOutput$!: Observable<number>;
  constructor() { }
  ...
}
```

2. We'll now combine all the streams to give out a single output—that is, the `outputStreamData` array. We'll remove all the existing `pipe` and `subscribe` functions from the `startStream` method, so the code should now look like this:

```
...
import { interval, merge, scan, Observable, takeWhile } from 'rxjs';
...
export class HomeComponent implements OnDestroy {
  ...
  startStream() {
    ...
    const fastestStreamSource = interval(500);
    this.streamsOutput$ = merge(
      streamSource,
      secondStreamSource,
      fastestStreamSource
    ).pipe(
      scan((acc, next) => {
        return [...acc, next];
      }, [] as number[])
    );
  }
  ...
}
```

3. Since we want to stop the stream on the **Stop Stream** button click, we'll use the `takeWhile` operator in the stream to work with the stream to only emit values when we click the **Start Stream** button and to stop when we hit the **Stop Stream** button. Update the `startStream` method in `home.component.ts` as follows:

```
startStream() {
    ...
    this.streamsOutput$ = merge(...).pipe(
      takeWhile(() => this.isStreamActive),
      scan((acc, next) => {
        return [...acc, next];
      }, [] as number[])
    )
  }
```

4. Remove the `ngOnDestroy` method since our stream is going to unsubscribe automatically when we leave the component (go to another route). This is because we're using the `async` pipe and Angular itself handles the subscription and unsubscription for us when using the `async pipe`. Also, we should remove the `implements OnDestroy` statement and remove the `OnDestroy` import.
5. Finally, modify the template in `home.component.html` to use the `streamOutput$` observable with the `async` pipe to loop over the output array:

```
    <div class="output-stream">
      <div class="input-stream__item" *ngFor="let item of
        streamsOutput$ | async">
        {{item}}
      </div>
    </div>
```

6. To verify that the subscription *really* gets destroyed on component destruction, let's put `console.log` in the `startStream` method inside the `tap` operator, as follows:

```
import { ..., takeWhile, tap } from 'rxjs';
startStream() {
    ...
    this.streamsOutput$ = merge(...).pipe(
      takeWhile(...),
      scan(...),
      tap((output) => console.log('output', output))
    )
  }
```

Hurray! With this change, you can try refreshing the app; navigate away from the **Home** route, and you'll see that the console logs stop as soon as you navigate away from the homepage. Also, you can start and stop the stream to see the output on the console. Do you feel great about what we just got by removing all that extra code? I certainly do. We'll see in the next section how it all works.

How it works...

Angular's `async` pipe automatically destroys/unsubscribes the subscription as soon as the component is destroyed. This gives us a great opportunity to use it where possible. In the recipe, we basically combined all the streams using the `merge` operator. The fun part was that for the `streamsOutput$` property, we wanted an observable of the output array on which we could loop over. However, merging the streams only combines them and emits the latest value emitted by any of the streams. So, we added a `pipe` function with the `scan` operator to take the latest output of the combined stream and add it to an array of all the previously emitted outputs. This sort of works like the `reduce` function of a JavaScript array.

Fun fact—streams don't emit any value unless they're subscribed to. "*But Ahsan, we didn't subscribe to the stream, we just merged and mapped the data. Where's the subscription?*" Glad you asked. Angular's `async` pipe subscribes to the stream itself, which triggers `console.log` as well, which we added in *Step 6* using the `tap` function.

IMPORTANT NOTE

The `async` pipe has a limitation, which is that you cannot stop the subscription until the component is destroyed. For cases where you want to subscribe and unsubscribe on a conditional basis, you'd want to go for operators such as the `takeWhile`/`takeUntil` operator or use a regular `unsubscribe` function yourself when the component is destroyed.

See also

- Angular `async` pipe documentation (`https://angular.io/api/common/AsyncPipe`)

Using the map operator to transform data

When making the API/HTTP calls in a web application, it is often the case that the server doesn't return the data in a form that is easy to directly render to the UI. We often need some sort of transformation of the data received from the server to map it to something our UI can work with. In this recipe, you're going to learn how to use the `map` operator to transform responses from an HTTP call.

Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-map-operator` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve rx-map-operator
```

This should open the app in a new browser tab, and you should see the following:

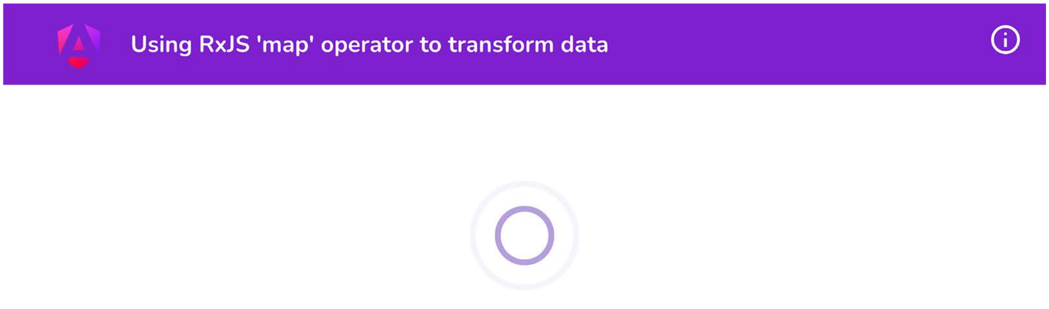


Figure 5.6: The rx-map-operator app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

Our template (`app.component.html`) for the app is set up already. And so is our `app.component.ts` file and the `appData` data structure that we require.

1. We'll start by creating a method in the `swapi.service.ts` file to fetch the data. We want only one function to be able to bring the data from different API calls, combine it, and return it. Update the file as follows:

```
...
import { delay, forkJoin, Observable } from 'rxjs';
import { IFilm, IPerson } from './interfaces';

...
export class SwapiService {
  ...
  fetchData(personId: string): Observable<{person:
    IPerson}> {

  }
  fetchPerson(id: string) {...}
  fetchPersonFilms(films: string[]) {...}
}
```

You're going to see that TypeScript is mad at us. Don't worry about it. We'll make it happy in time.

2. Let's add the following code to first get the person, and then get the films of that person in the `fetchData` function:

```
...
export class SwapiService {
  ...

  fetchData(personId: string): Observable<{person: IPerson}> {
    let personInfo: IPerson;
    return this.fetchPerson(personId)
      .pipe(
        mergeMap((person) => {
          personInfo = person;
            return this.fetchPersonFilms(person.films);
        })
      )
  }
  ...
}
```

   Now we can decide what to do when we receive the films back.

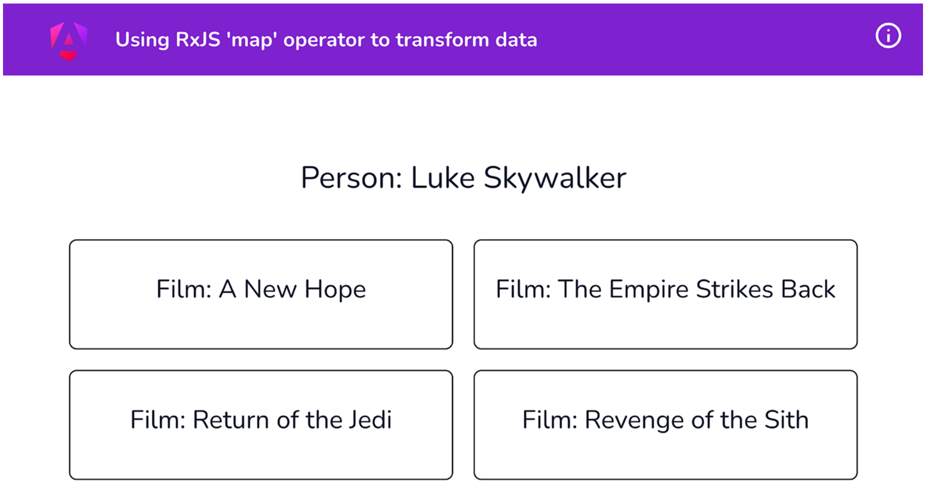
3. We'll map over the response from the `films` HTTP calls and add that to the `personInfo` object. Update the `swapi.service.ts` file as follows:

```
...
import { delay, forkJoin, map, mergeMap, Observable } from 'rxjs';
...
export class SwapiService {
 ...

  fetchData(personId: string): Observable<{ person: IPerson }> {
    let personInfo: IPerson;
    return this.fetchPerson(personId).pipe(
      mergeMap((person) => {
        personInfo = person;
        return this.fetchPersonFilms(person.films);
      }),
      map((films: IFilm[]) => {
        personInfo.filmObjects = films;
        return {
          person: personInfo,
        };
      })
    );
  }
  ...
}
```

4. Finally, let's use the `fetchData` method from `SwapiService` inside our `app.component.ts` file. Update the `fetchData` method in the file as follows and make sure you remove unused dependencies from the file:

```
...
export class AppComponent implements OnInit {
  ...
  fetchData() {
    this.loadingData = true;
    this.swapi.fetchData('1').subscribe((response) => {
      this.appData = response;
      this.loadingData = false;
    });
  }
}
```

And yes! If you now refresh the app, you'll notice that the data is being displayed in the view:



Using RxJS 'map' operator to transform data

Person: Luke Skywalker

Film: A New Hope

Film: The Empire Strikes Back

Film: Return of the Jedi

Film: Revenge of the Sith

Figure 5.7: The UI showing the received data from swapi

Now that you've finished the recipe, see the next section on how this works.

How it works...

The `map` operator is one of the most used RxJS operators of all time. Especially in Angular when we make HTTP calls. In this recipe, our target was to do as little work as possible in the `app.component.ts` file. That's because as one of the community-adopted practices, the component should request data from the service and the service should give it in such a way that it can be bound to the UI variables as is. The Angular documentation also encourages keeping the component's code as tiny as possible. It is also generally a good idea to distribute your code into different layers, i.e., in components, services, pipes, etc. This is to be able to scale the application with ease, have better testing possibilities, and be able to easily replace the layers with a completely different thing. Therefore, we created this `fetchData` method in the `SwapiService` class to use the `fetchPerson` and `fetchPersonFilms` methods to first make the HTTP calls, and then we used the `map` operator to transform the data into exactly the data structure our component/UI is expecting.

See also

- `map operator documentation` (`https://www.learnrxjs.io/learn-rxjs/operators/transformation/map`)

Using the switchMap and debounceTime operators with autocompletes for better performance

For a lot of apps, we have features such as searching content as the user types. This is a really good **User Experience** (**UX**) as the user doesn't have to press a button to do a search. However, if we send an HTTP call to the server on every key press, that's going to result in a lot of HTTP calls being sent, and we can't know which HTTP call will complete first; thus, we can't be sure if we will have the correct data shown in the view or not. In this recipe, you'll learn how to use the `switchMap` operator to cancel out the last subscription and create a new one instead. This would result in canceling previous HTTP calls and keeping only one HTTP call—the last one. We'll use the `debounceTime` operator to wait for the input to be idle before it even tries to make one call.

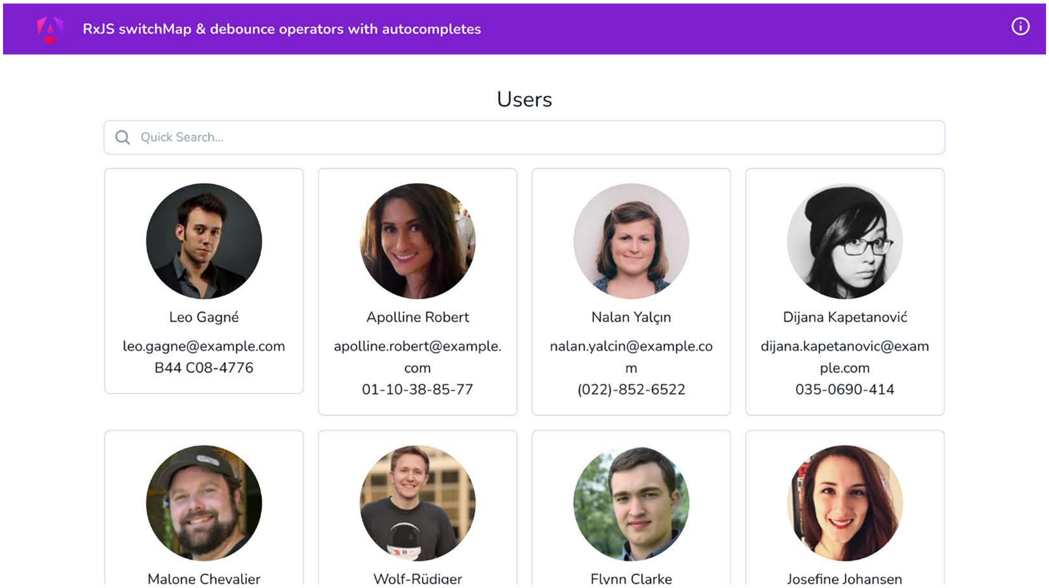
Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-switchmap-operator` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve rx-switchmap-operato
```

This should open the app in a new browser tab, and you should see the following:



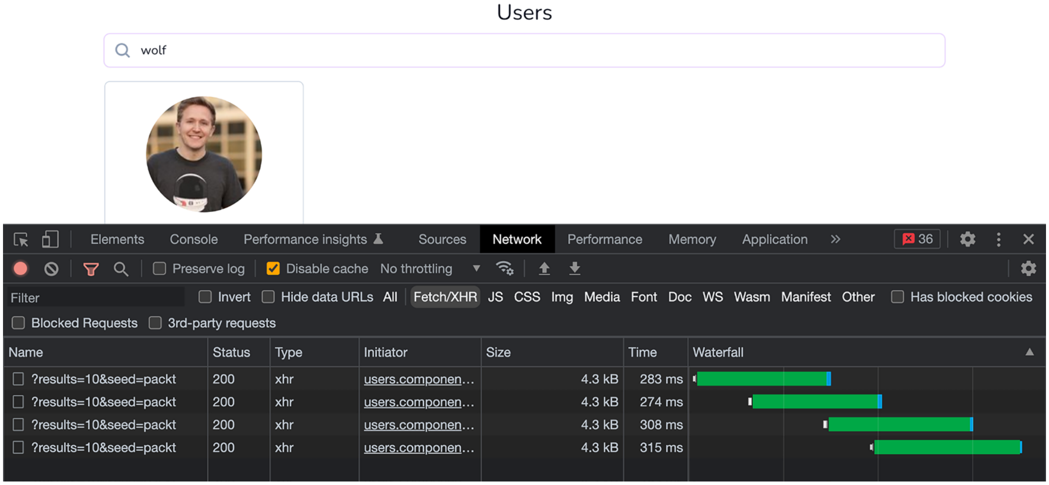
RxJS switchMap & debounce operators with autocompletes
Quick Search...

Users

Leo Gagné leo.gagne@example.com B44 C08-4776	Apolline Robert apolline.robert@example.com 01-10-38-85-77	Nalan Yalçın nalan.yalcin@example.com (022)-852-6522	Dijana Kapetanović dijana.kapetanovic@example.com 035-0690-414
Malone Chevalier	Wolf-Rüdiger	Flynn Clarke	Josefine Johansen

Figure 5.8: The rx-switchmap-operator app running on http://localhost:4200

Now that we have the app running locally, open **Chrome DevTools** and go to the **Network** tab. Type `wolf` in the search input, and you'll see four calls being sent to the API server, as follows:



Users
wolf
Elements
Console
Performance insights
Sources
Network
Performance
Memory
Application
36
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
Has blocked cookies
Blocked Requests
3rd-party requests

Name	Status	Type	Initiator	Size	Time	Waterfall
?results=10&seed=packt	200	xhr	users.componen…	4.3 kB	283 ms	
?results=10&seed=packt	200	xhr	users.componen…	4.3 kB	274 ms	
?results=10&seed=packt	200	xhr	users.componen…	4.3 kB	308 ms	
?results=10&seed=packt	200	xhr	users.componen…	4.3 kB	315 ms	

Figure 5.9: A separate HTTP call is sent for each input change

How to do it…

You can start typing into the search box on the homepage to see the filtered users, and if you see the **Network** tab, you'll notice that whenever the input changes, we send a new HTTP call. Let's avoid sending a call on each keypress by using the `switchMap` operator.

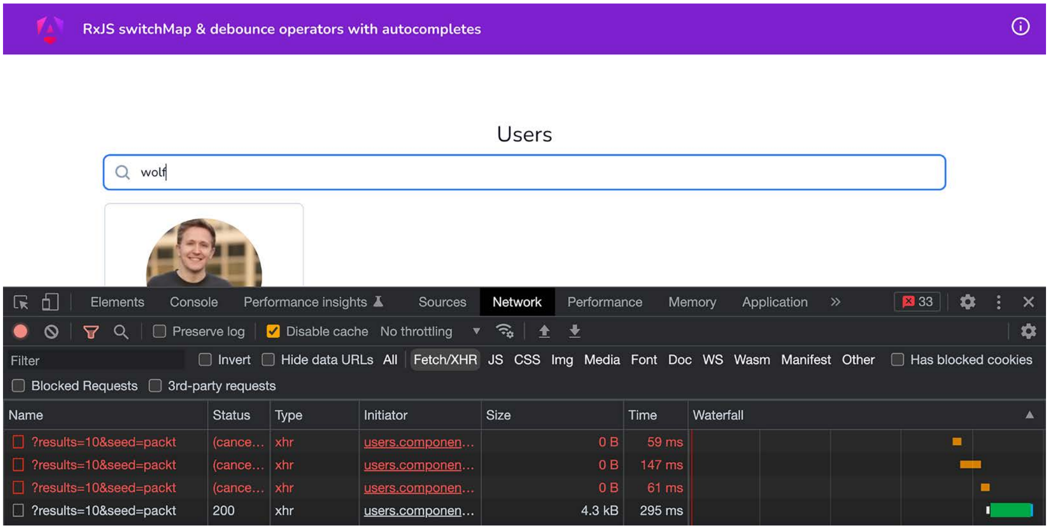
1. First, import the `switchMap` operator from `rxjs/operators` in the `users/users.component.ts` file, as follows:

   ```
   ...
   import { mergeMap, startWith, takeWhile, switchMap } from 'rxjs/
   operators';
   ```

2. We will now modify our subscription to the `username` form control— specifically, the `valueChanges` observable to use the `switchMap` operator for the `this.userService.searchUsers(query)` method call. This returns an `Observable` containing the result of the HTTP call. The code should look like this:

   ```
   ngOnInit() {
       ...
       this.searchForm.controls['username'].valueChanges
         .pipe(
           startWith(''),
           takeWhile(() => this.componentAlive),
           switchMap((query) =>
             this.userService.searchUsers(query))
         )
         .subscribe((users) => {...});
     }
   ```

   If you refresh the app now, open **Chrome DevTools**, and check the network type while typing `wolf` really fast, you'll see that all the previous calls are canceled and we only have the latest HTTP call succeeding:


RxJS switchMap & debounce operators with autocompletes
Users
wolf
Elements
Console
Performance insights
Sources
Network
Performance
Memory
Application
33
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
Has blocked cookies
Blocked Requests
3rd-party requests

Name	Status	Type	Initiator	Size	Time	Waterfall
?results=10&seed=packt	(cance…	xhr	users.componen…	0 B	59 ms	
?results=10&seed=packt	(cance…	xhr	users.componen…	0 B	147 ms	
?results=10&seed=packt	(cance…	xhr	users.componen…	0 B	61 ms	
?results=10&seed=packt	200	xhr	users.componen…	4.3 kB	295 ms	

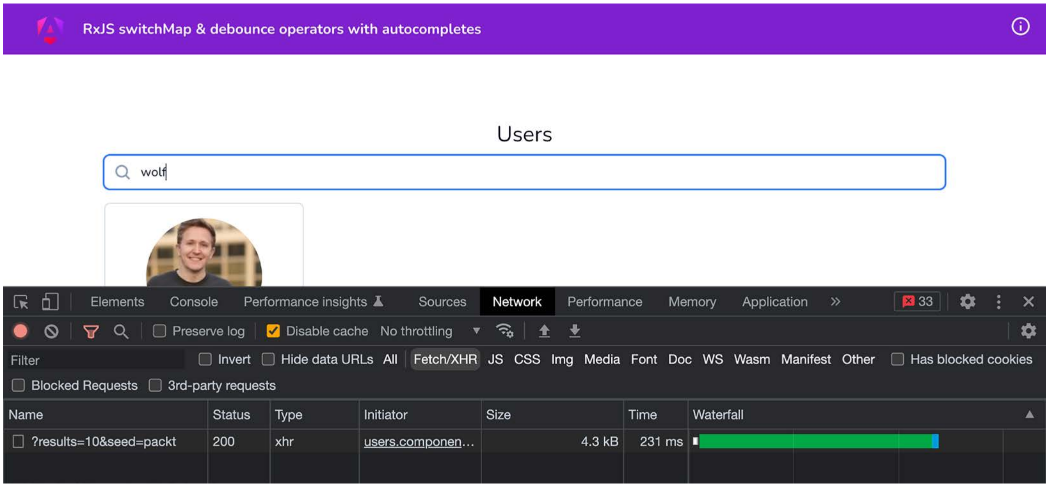
Figure 5.10: switchMap canceling prior HTTP calls

Well, this looks better, but the `backend/api` endpoint still receives those calls.

3. We're going to use the `debounceTime` operator now to wait for the search input to be idle before starting to execute a call. Update the `users.component.ts` file as follows:

```
...
import { debounceTime, ...} from 'rxjs/operators';
...
export class UsersComponent implements OnInit {
  ...
  ngOnInit() {
    ...
    this.searchForm.controls['username'].valueChanges
      .pipe(
        startWith(''),
        debounceTime(500),
        takeWhile(() => this.componentAlive),
        switchMap((query) =>
          this.userService.searchUsers(query))
      )
      .subscribe((users) => {...});
  }
}
```

Figure 5.11 shows that there is only one call sent to the server, even after typing four letters in the search input:



RxJS switchMap & debounce operators with autocompletes

Users

wolf

Name	Status	Type	Initiator	Size	Time	Waterfall
☐ ?results=10&seed=packt	200	xhr	users.componen…	4.3 kB	231 ms	

Elements
Console
Performance insights
Sources
Network
Performance
Memory
Application
33
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
Has blocked cookies
Blocked Requests
3rd-party requests

Figure 5.11: debounceTime waiting for the input to be idle

Woot! We now have only one call that'll succeed, process the data, and end up in the view; see the next section on how it works.

How it works...

The `switchMap` operator cancels the previous (inner) subscription and subscribes to a new observable instead. In our case, the parent observable (the input element's `valueChanges` emitter) emits a value, and the `switchMap` operator cancels the previous operation in progress. That's why it cancels all the HTTP calls sent before in our example and just subscribes to the last one. However, the call still reaches the API endpoint. If this was our server, we may still receive the API calls, so we use the `debounceTime` operator on the form control to wait for the input to be idle (for 500 ms) before we even send our first call.

See also

- `switchMap` operator documentation (`https://www.learnrxjs.io/learn-rxjs/operators/transformation/switchmap`)
- `debounceTime` operator documentation (`https://www.learnrxjs.io/learn-rxjs/operators/filtering/debouncetime`)

Creating a custom RxJS operator

By following the other recipes in this chapter, I have to ask if you've become a fan of RxJS yet? *Have you?* Well, I am. And in this recipe, you're going to level up your RxJS game. You're going to create your own custom RxJS operator that just taps into any observable stream and logs the values on the console. We'll call it the `logWithLabel` operator.

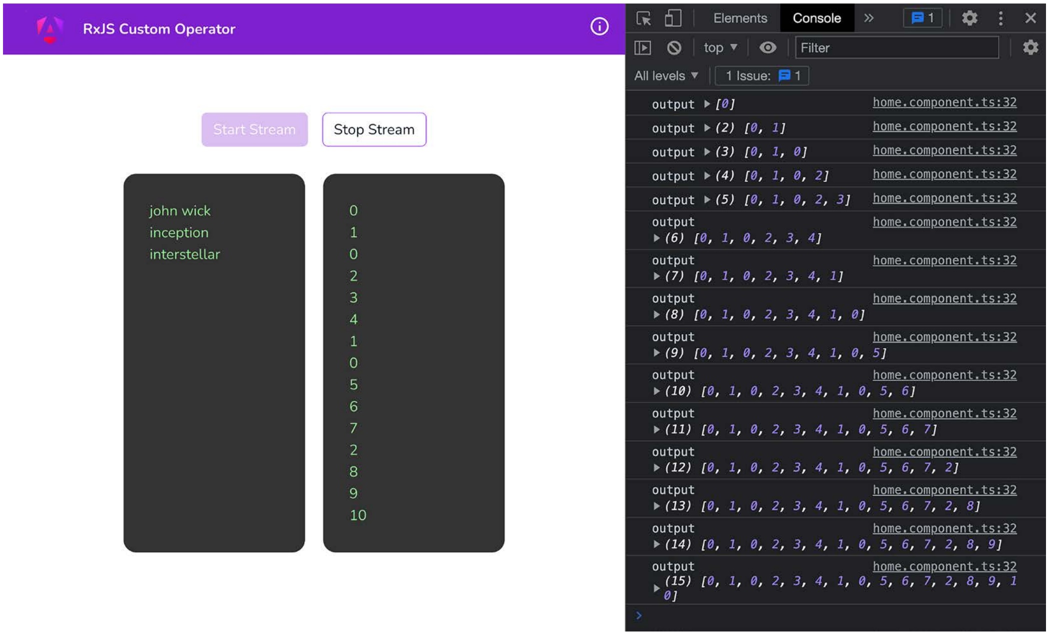
Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-custom-operator` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve rx-custom-operator
```

This should open the app in a new browser tab. If you click the **Start Stream** button while you have the DevTools open, you should see the following:



```
john wick
inception
interstellar
```

```
0
1
0
2
3
4
1
0
5
6
7
2
8
9
10
```

```
output ▸[0]                                                   home.component.ts:32
output ▸(2) [0, 1]                                            home.component.ts:32
output ▸(3) [0, 1, 0]                                         home.component.ts:32
output ▸(4) [0, 1, 0, 2]                                      home.component.ts:32
output ▸(5) [0, 1, 0, 2, 3]                                   home.component.ts:32
output                                                        home.component.ts:32
 ▸(6) [0, 1, 0, 2, 3, 4]
output                                                        home.component.ts:32
 ▸(7) [0, 1, 0, 2, 3, 4, 1]
output                                                        home.component.ts:32
 ▸(8) [0, 1, 0, 2, 3, 4, 1, 0]
output                                                        home.component.ts:32
 ▸(9) [0, 1, 0, 2, 3, 4, 1, 0, 5]
output                                                        home.component.ts:32
 ▸(10) [0, 1, 0, 2, 3, 4, 1, 0, 5, 6]
output                                                        home.component.ts:32
 ▸(11) [0, 1, 0, 2, 3, 4, 1, 0, 5, 6, 7]
output                                                        home.component.ts:32
 ▸(12) [0, 1, 0, 2, 3, 4, 1, 0, 5, 6, 7, 2]
output                                                        home.component.ts:32
 ▸(13) [0, 1, 0, 2, 3, 4, 1, 0, 5, 6, 7, 2, 8]
output                                                        home.component.ts:32
 ▸(14) [0, 1, 0, 2, 3, 4, 1, 0, 5, 6, 7, 2, 8, 9]
output                                                        home.component.ts:32
 ▸(15) [0, 1, 0, 2, 3, 4, 1, 0, 5, 6, 7, 2, 8, 9, 1
0]
>
```

Figure 5.12: The rx-custom-operator app running on http://localhost.4200

Let's jump into the recipe steps in the next section.

How to do it…

We're going to create a custom RxJS operator named `logWithLabel`, which will log the values of the observable stream on the console with a label.

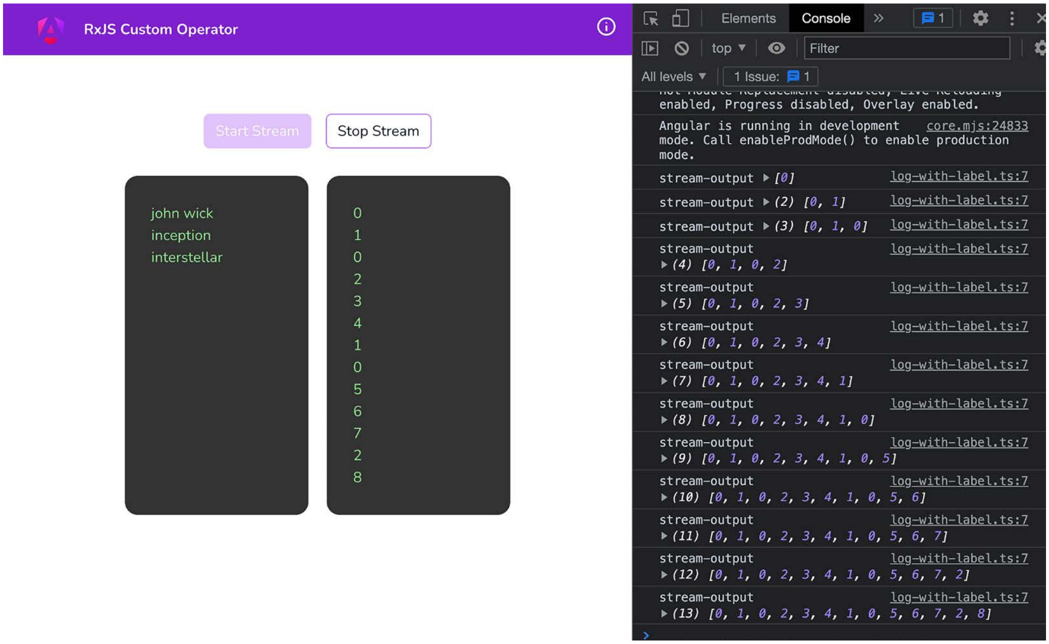
1. Create a new file inside the app folder and name it `log-with-label.ts`. Then add the following code inside the file:

```
import { Observable } from 'rxjs/internal/Observable';
import { tap } from 'rxjs/operators';



const logWithLabel = <T>(
  label: string
): ((source$: Observable<T>) => Observable<T>) => {
  return (source$) => source$.pipe(tap((value) =>
    console.log(label, value)));
};

export default logWithLabel;
```

2. Now we can import the `logWithLabel` operator from the `log-with-label.ts` file inside the `home/home.component.ts` file, as follows:

```
...
import logWithLabel from '../log-with-label';
@Component({...})
export class HomeComponent {
  ...
  startStream() {
    ...
    this.streamsOutput$ = merge(...).pipe(
      takeWhile(...),
      scan(...),
      logWithLabel('stream-output')
    );
  }
  ...
}
```

And that's it! If you refresh the app and click the **Start Stream** button, you can see the output using the `logWithLabel` operator, as follows:


RxJS Custom Operator
Start Stream
Stop Stream
john wick
inception
interstellar
0
1
0
2
3
4
1
0
5
6
7
2
8
Elements
Console
1
top
Filter
All levels
1 Issue: 1

```
enabled, Progress disabled, Overlay enabled.
Angular is running in development    core.mjs:24833
mode. Call enableProdMode() to enable production
mode.
stream-output ▶ [0]                  log-with-label.ts:7
stream-output ▶ (2) [0, 1]           log-with-label.ts:7
stream-output ▶ (3) [0, 1, 0]        log-with-label.ts:7
stream-output                        log-with-label.ts:7
▶ (4) [0, 1, 0, 2]
stream-output                        log-with-label.ts:7
▶ (5) [0, 1, 0, 2, 3]
stream-output                        log-with-label.ts:7
▶ (6) [0, 1, 0, 2, 3, 4]
stream-output                        log-with-label.ts:7
▶ (7) [0, 1, 0, 2, 3, 4, 1]
stream-output                        log-with-label.ts:7
▶ (8) [0, 1, 0, 2, 3, 4, 1, 0]
stream-output                        log-with-label.ts:7
▶ (9) [0, 1, 0, 2, 3, 4, 1, 0, 5]
stream-output                        log-with-label.ts:7
▶ (10) [0, 1, 0, 2, 3, 4, 1, 0, 5, 6]
stream-output                        log-with-label.ts:7
▶ (11) [0, 1, 0, 2, 3, 4, 1, 0, 5, 6, 7]
stream-output                        log-with-label.ts:7
▶ (12) [0, 1, 0, 2, 3, 4, 1, 0, 5, 6, 7, 2]
stream-output                        log-with-label.ts:7
▶ (13) [0, 1, 0, 2, 3, 4, 1, 0, 5, 6, 7, 2, 8]
>
```

Figure 5.13: Logs using the logWithLabel custom RxJS operator

See the next section to understand how it all works.

How it works...

A custom RxJS operator is a function that should take an observable source stream and return *something*. That *something* is usually an observable. In this recipe, we wanted to tap into the stream to log something on the console every time the stream emits a value. We also wanted to have a custom label for the logs for this stream. This is why we ended up creating the custom operator as a `factory` function that can take `label` as input, i.e., when we call the `logWithLabel` function (let's call it `function A`), it returns a function (let's call it `function B`) from within. The returned function (`B`) is what RxJS calls with the observable stream when we use the `logWithLabel` method inside the `pipe` function of the stream. Inside `function B`, we use the `tap` operator from RxJS to intercept the source observable and to log the values on the console using the `label` provided.

See also

- `tap` operator documentation (`https://rxjs.dev/api/operators/tap`)
- `Observable` documentation (`https://rxjs.dev/guide/observable`)

Retrying failed HTTP calls with RxJS

In this recipe, you're going to learn how to retry HTTP calls smartly with RxJS operators. We're going to use a technique called the **exponential backoff** technique. This means that we retry the HTTP calls but with each next call having a delay more than the previous time for the attempt, and we stop after several maximum tries. Sounds exciting? Let's get into it.

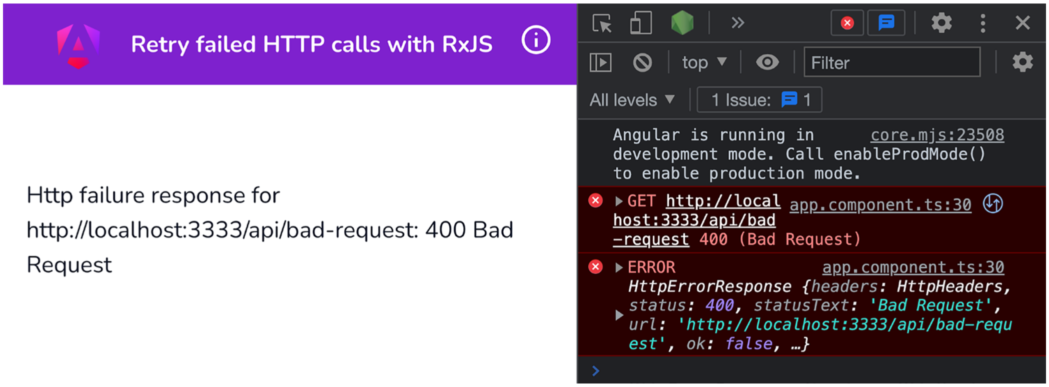
Getting ready

The app that we are going to work with resides in `start/apps/chapter05/rx-retry-http-calls` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project with the backend server:

```
npm run serve rx-retry-http-calls with-server
```

This should open the app in a new browser tab, and you should see the following:



Http failure response for http://localhost:3333/api/bad-request: 400 Bad Request

```
Angular is running in        core.mjs:23508
development mode. Call enableProdMode()
to enable production mode.
▶GET http://local app.component.ts:30
host:3333/api/bad
-request 400 (Bad Request)
▶ERROR                    app.component.ts:30
 HttpErrorResponse {headers: HttpHeaders,
 status: 400, statusText: 'Bad Request',
▶url: 'http://localhost:3333/api/bad-requ
 est', ok: false, …}
```

Figure 5.14: The rx-retry-http-calls running on http://localhost.4200

Let's jump into the recipe steps in the next section.

How to do it...

We're going to create a custom RxJS operator named `backoff` that will retry the HTTP calls for us with the exponential `backoff` strategy.

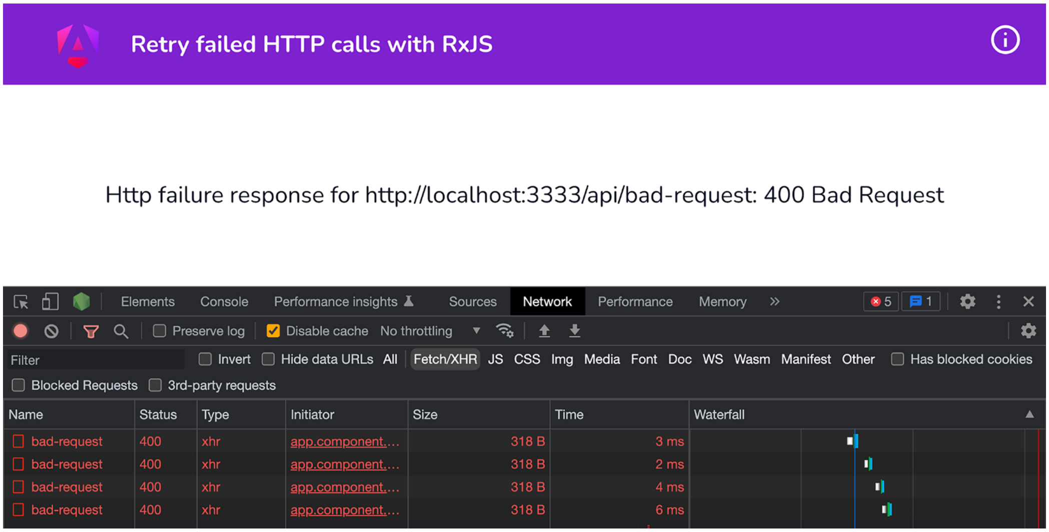
1. Create a new file inside the app folder and name it `retry-backoff.ts`. Then add the following code inside the file:

```
import { of, pipe, throwError } from 'rxjs';
import { retry } from 'rxjs/operators';
export function retryBackoff(maxTries: number, delay: number) {
  return pipe(
    retry({
      delay: (error, retryCount) => {
        return retryCount > maxTries ? throwError(() =>
          error) : of(retryCount);
      },
    })
  );
}
```

2. Now let's use this operator in `app.component.ts` to retry the HTTP calls. Update the file as follows:

```
...
import { retryBackoff } from './retry-backoff';
...
export class AppComponent implements OnInit {
  ...
  ngOnInit(): void {
    this.isMakingHttpCall = true;
    this.http
      .get('http://localhost:3333/api/bad-request')
      .pipe(
        retryBackoff(3, 300),
        catchError(...)
      )
      .subscribe(...);
  }
}
```

If you refresh the app, you'll notice that now we retry the HTTP calls. But all of the retries are done instantly (notice the **Waterfall** column), as shown in *Figure 5.15*. We don't want that. We want every next try to be done with an increasing delay.


Retry failed HTTP calls with RxJS
Elements
Console
Performance insights
Sources
Network
Performance
Memory
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
Has blocked cookies
Blocked Requests
3rd-party requests

Http failure response for http://localhost:3333/api/bad-request: 400 Bad Request

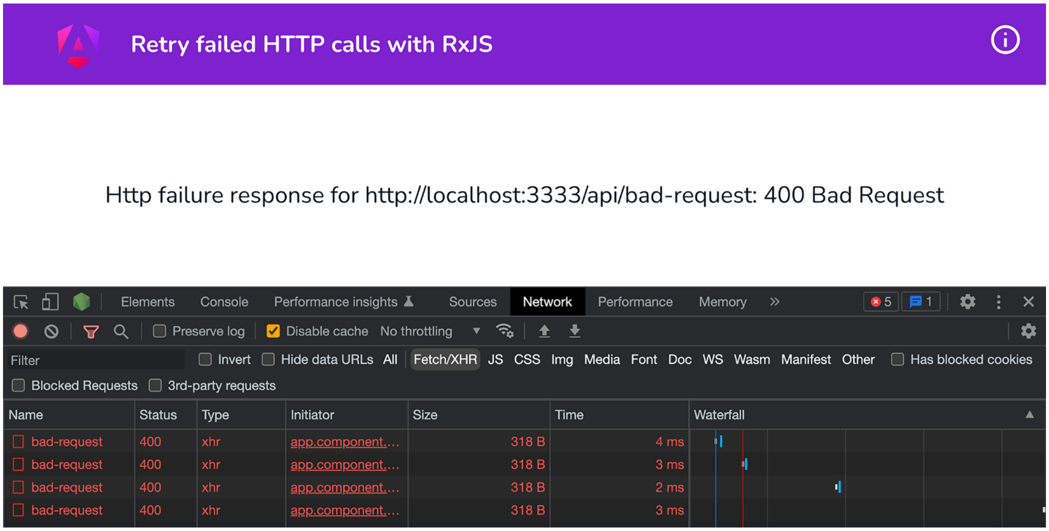
Name	Status	Type	Initiator	Size	Time	Waterfall
bad-request	400	xhr	app.component.…	318 B	3 ms	
bad-request	400	xhr	app.component.…	318 B	2 ms	
bad-request	400	xhr	app.component.…	318 B	4 ms	
bad-request	400	xhr	app.component.…	318 B	6 ms	

Figure 5.15: Retrying HTTP calls multiple times instantly

3. Update the `retry-backoff.ts` file to add a delay using the `timer` operator and some calculations, as follows:

```
import { of, pipe, throwError, timer } from 'rxjs';
import { map, mergeMap, retry } from 'rxjs/operators';

export function retryBackoff(maxTries: number, delay: number) {
  return pipe(
    retry({
      delay: (error, retryCount) => {
        return (
          retryCount > maxTries ? throwError(() => error) :
            of(retryCount)
        ).pipe(
          map((count) => count * count),
          mergeMap((countSq) => timer(countSq * delay))
        );
      },
    })
  );
}
```

And that's it! If you refresh the app, you'll see that every next retry of the HTTP call has an increased delay compared to the previous one. Notice how far off the last HTTP call is in the **Waterfall** column (it is on the right edge of *Figure 5.16*):


Retry failed HTTP calls with RxJS

Http failure response for http://localhost:3333/api/bad-request: 400 Bad Request

Elements
Console
Performance insights
Sources
Network
Performance
Memory
5
1
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
Has blocked cookies
Blocked Requests
3rd-party requests

Name	Status	Type	Initiator	Size	Time	Waterfall
bad-request	400	xhr	app.component.…	318 B	4 ms	
bad-request	400	xhr	app.component.…	318 B	3 ms	
bad-request	400	xhr	app.component.…	318 B	2 ms	
bad-request	400	xhr	app.component.…	318 B	3 ms	

Figure 5.16: Retrying HTTP calls with exponential backoff

See the next section to understand how it all works.

How it works…

The `retry` operator has two overloads (at the time of writing this book). One of them takes `number` and RxJS will just have the observable retried the number of times equal to `number` (until it throws an exception). The other overload is that it takes a configuration object. In the configuration object, we're using the `delay` function to handle our logic. The `delay` function receives `error` and the `retryCount` from RxJS, which we use to throw the error if we have already done our maximum tries or to pass forward `retryCount`. We get the maximum number of tries from our `retryBackoff` function's arguments. Finally, we make the `map` and `mergeMap` opeartors work with `delay`. Using the `map` operator, we take a square of the `retryCount` variable's value. And then, in the `mergeMap` operator, we multiply the square value with the delay provided to the `retryBackoff` function. As a result, every next request takes the delay equal to `((retryCount * retryCount) * delay)`. Notice that we're using the `timer` function to have RxJS wait before it can retry the HTTP call again.

See also

- **RxJS custom operators** (`https://indepth.dev/posts/1421/rxjs-custom-operators`)
- **Exponential backoff** documentation (`https://angular.io/guide/practical-observable-usage#exponential-backoff`)

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

6

Reactive State Management with NgRx

Angular and **reactive programming** are best buddies, and handling an app's state reactively is one of the best things you can do with your app. **NgRx** (short for **Angular Reactive Extensions**) is a framework that provides a set of libraries as reactive extensions for Angular. In this chapter, you'll learn how to use the NgRx ecosystem to manage your app's state reactively, and you'll also learn about a couple of cool things the NgRx ecosystem will help you with.

Here are the recipes we're going to cover in this chapter:

- Creating your first NgRx store with actions and reducers
- Using NgRx Store Devtools to debug state changes
- Using NgRx selectors to select and render state in components
- Using NgRx effects to fetch data from API calls
- Using NgRx Component Store to manage the state of a component

Technical requirements

For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter06`.

Creating your first NgRx store with actions and reducers

In this recipe, you'll work your way through understanding NgRx's basics by setting up your first NgRx store. You'll also create some actions, along with a reducer, and to see the changes in the reducer, we'll be putting in appropriate console logs.

Getting ready

The app that we are going to work with resides in `start/apps/chapter06/ngrx-actions-reducer` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ngrx-actions-reducer
```

This should open the app in a new browser tab. If you add a couple of items, you should see the following:

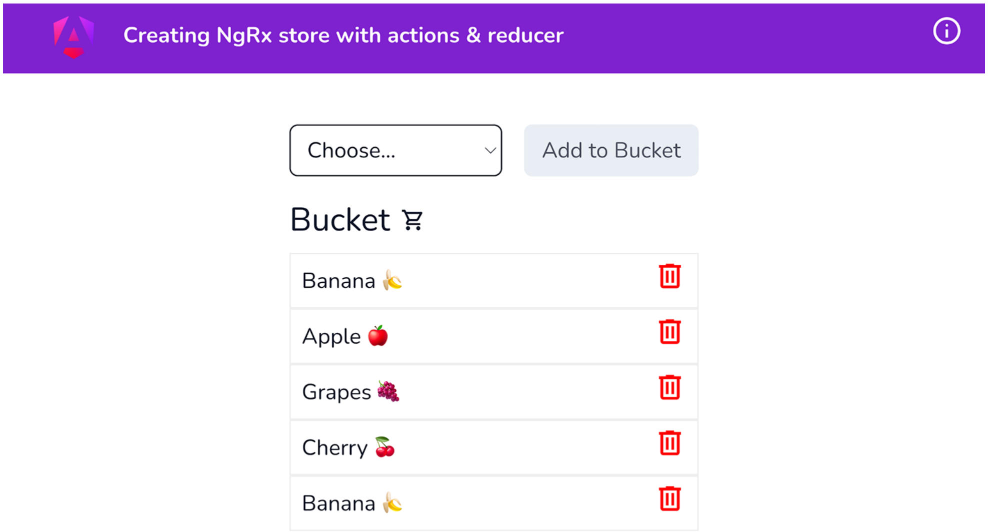


Figure 6.1: ngrx-actions-reducers app running on http://localhost:4200

Now that we have the app running, we'll move on to the steps for the recipe.

How to do it...

We have an Angular application with a single page containing a bucket. You can add fruits to your bucket and remove items from the bucket. We already have the `@ngrx/store` package installed in the workspace, so you don't have to install it. However, when you work on a standalone app (or on some of your own projects), you would start by adding NgRx and running the following command:

```
npm install @ngrx/store
```

Update the `app.config.ts` file to provide the NgRx store as follows:

```
...
import { provideStore } from '@ngrx/store';
...
import { provideAnimations } from '@angular/platform-browser/animations';
import { provideStore } from '@ngrx/store';
export const appConfig: ApplicationConfig = {
  providers: [
    ...,
    provideAnimations(),
    provideStore({}),
  ],
};
```

Notice that we've passed an empty object `{}` to the `provideStore()` method; we'll change that going forward.

Now, we'll create some actions. Create a folder named `store` inside the app folder. Then, create a file named `bucket.actions.ts` inside the `store` folder, and finally, add the following code to the newly created file:

```
import { createActionGroup, props } from '@ngrx/store';
import { IFruit } from '../interfaces/fruit.interface';

export const BucketActions = createActionGroup({
  source: 'Bucket',
  events: {
    'Add Fruit': props<{ fruit: IFruit }>(),
    'Remove Fruit': props<{ fruitId: number }>(),
  },
});
```

Since we have the actions in place now, we must create a reducer.

Create a new file inside the `store` folder, name it `bucket.reducer.ts`, and add the following code to it to define the necessary imports and the initial state:

```
import { createReducer, on } from '@ngrx/store';
import { IFruit } from '../interfaces/fruit.interface';
import { BucketActions } from './bucket.actions';

export const initialState: ReadonlyArray<IFruit> = [];
```

Now we'll define the reducer. Add the following code to the `bucket.reducer.ts` file:

```
...
export const bucketReducer = createReducer(
  initialState,
  on(BucketActions.addFruit, (_state, { fruit }) => {
    console.log({ fruit });
    return [fruit, ..._state];
  }),
  on(BucketActions.removeFruit, (_state, { fruitId }) => {
    console.log({ fruitId });
    return _state.filter((fr) => fr.id === fruitId);
  })
);
```

Next, we'll update the `app.config.ts` file to use the reducer we just created. Update the file as follows:

```
...
import { bucketReducer } from './app/store/bucket.reducer';

bootstrapApplication(AppComponent, {
  providers: [
    ...,
    provideStore({
      bucket: bucketReducer,
    }),
  ],
}).catch((err) => console.error(err));
```

Now we will use the actions we created to see the console logs from the reducer functions. Update the `bucket.component.ts` file to first use the NgRx store as follows:

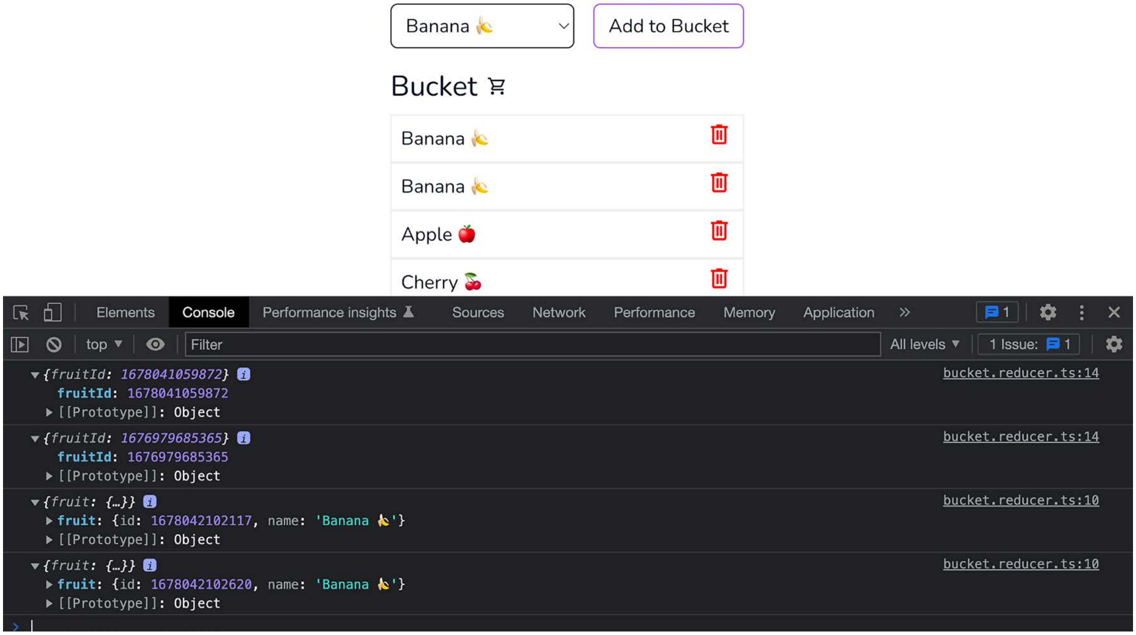
```
...
import { StoreModule, Store } from '@ngrx/store';
@Component({
  ...,
  imports: [CommonModule, FormsModule, StoreModule],
})
export class BucketComponent implements OnInit {
  ...
  fruits: string[] = Object.values(Fruit);
  store = inject(Store);
  ...
}
```

Let's dispatch actions to the store for when we're adding fruits to and removing fruits from the bucket. We'll also modify the code a bit to avoid repetition. Update the `bucket.component.ts` file as follows:

```
...
import { BucketActions } from '../store/bucket.actions';
...
export class BucketComponent implements OnInit {
  ...
    addSelectedFruitToBucket() {
```

```
    const newFruit: IFruit = {
      id: Date.now(),
      name: this.selectedFruit,
    };
    this.store.dispatch(
      BucketActions.addFruit({
        fruit: newFruit,
      })
    );
    this.bucketService.addItem(newFruit);
  }
  deleteFromBucket(fruit: IFruit) {
    this.store.dispatch(
      BucketActions.removeFruit({
        fruitId: fruit.id,
      })
    );
    this.bucketService.removeItem(fruit);
  }
}
```

Now if you add or remove items from the bucket, you're going to see the logs on the console, as follows in *Figure 6.2*, which means our actions and reducer are working:


Banana 🍌
Add to Bucket
Bucket 🛒
Banana 🍌
Banana 🍌
Apple 🍎
Cherry 🍒
Elements
Console
Performance insights
Sources
Network
Performance
Memory
Application
top
Filter
All levels
1 Issue: 1

```
▼{fruitId: 1678041059872} ℹ                                       bucket.reducer.ts:14
   fruitId: 1678041059872
  ▶[[Prototype]]: Object
▼{fruitId: 1676979685365} ℹ                                       bucket.reducer.ts:14
   fruitId: 1676979685365
  ▶[[Prototype]]: Object
▼{fruit: {…}} ℹ                                                   bucket.reducer.ts:10
  ▶fruit: {id: 1678042102117, name: 'Banana 🍌'}
  ▶[[Prototype]]: Object
▼{fruit: {…}} ℹ                                                   bucket.reducer.ts:10
  ▶fruit: {id: 1678042102620, name: 'Banana 🍌'}
  ▶[[Prototype]]: Object
```

Figure 6.2: Logs showing the actions for adding and removing items from a bucket

And that covers it all for this recipe! You are now able to integrate an NgRx store into an Angular app, create NgRx actions, and dispatch those actions. You're also able to create a reducer, define its state, and listen to actions to act on them.

See also

- NgRx store walkthrough tutorial (`https://ngrx.io/guide/store/walkthrough`)
- NgRx reducers documentation (`https://ngrx.io/guide/store/reducers`)
- NgRx actions documentation (`https://ngrx.io/guide/store/actions`)

Using NgRx Store Devtools to debug state changes

In this recipe, you'll learn how to use `@ngrx/store-devtools` to debug your app's state, the dispatched actions, and the difference in the state when the actions are dispatched. We'll be using an existing app we're familiar with to learn about the process.

Getting ready

The app that we are going to work with resides in `start/apps/chapter06/ngrx-devtools` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ngrx-devtools
```

This should open the app in a new browser tab. If you add a couple of items, you should see the following:

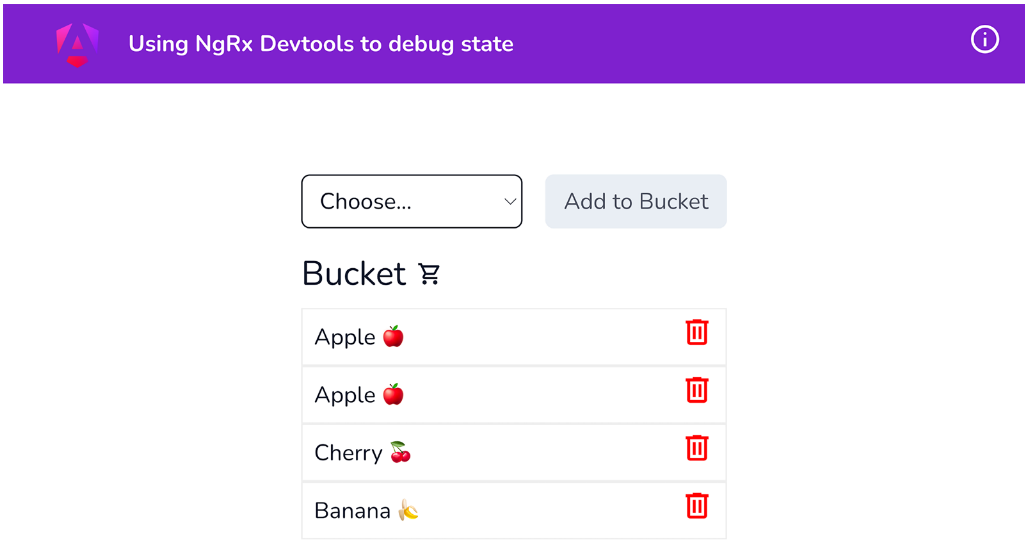


Figure 6.3: Using the ngrx-devtools app running on http://localhost:4200

Now that we have the app set up, let's see the steps of the recipe in the next section.

How to do it...

We have an Angular app that already has the `@ngrx/store` package integrated. We also have a reducer set up and some actions in place that are logged on the console as soon as you add or remove an item. We already have the `@ngrx/store-devtools` package installed in the workspace, so you don't have to install it. However, when you work on a standalone app (or on some of your own projects), you would start by adding NgRx store `devtools` and running the following command:

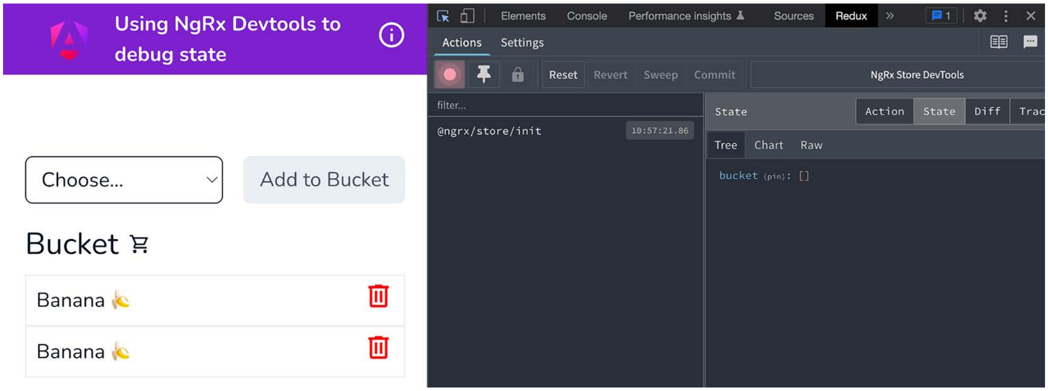
```
npm install @ngrx/store-devtools
```

1. First, update your `app.config.ts` file to include a `StoreDevtoolsModule.instrument` entry, as follows:

```
...
import { provideStoreDevtools } from '@ngrx/store-devtools';
import { bucketReducer } from './app/store/bucket.reducer';

bootstrapApplication(AppComponent, {
  providers: [
    ...,
    provideStore({
      bucket: bucketReducer,
    }),
    provideStoreDevtools({
      maxAge: 50,
    }),
  ],
}).catch((err) => console.error(err));
```

2. And now download the **Redux DevTools** extension from `https://github.com/zalmoxisus/redux-devtools-extension/` for your browser and install it. I'll be consistently using the Chrome browser in this book.

3. Open **Chrome DevTools**. There should be a new tab named **Redux**. Tap it and refresh the page. You'll see something like this:

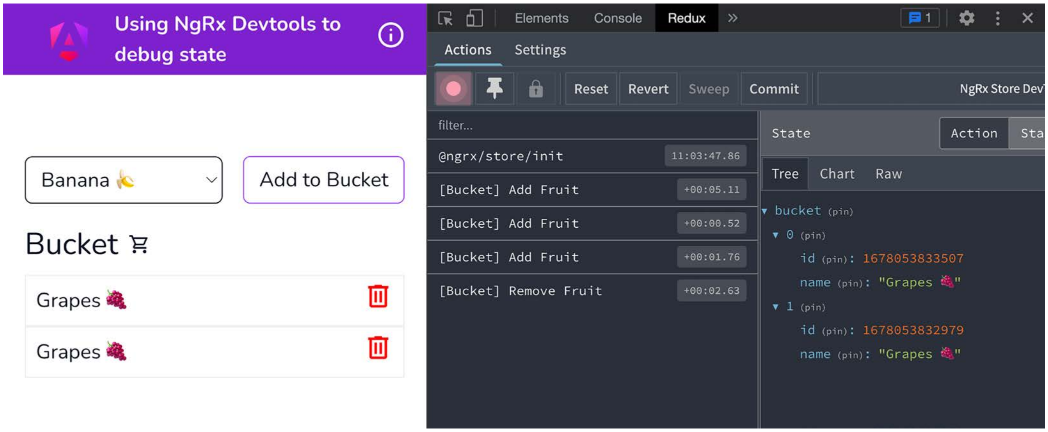

Using NgRx Devtools to debug state
Choose...
Add to Bucket
Bucket
Banana 🍌
Banana 🍌
Elements
Console
Performance insights
Sources
Redux
Actions
Settings
Reset
Revert
Sweep
Commit
NgRx Store DevTools
filter...
State
Action
State
Diff
Trac
Tree
Chart
Raw

```
@ngrx/store/init                10:57:21.86
```

```
bucket (pin): []
```

Figure 6.4: Redux DevTools showing the initial Redux action dispatched

4. Remove all the fruits, add two cherries and a banana to the bucket, and then remove the banana from the bucket. You should see all the relevant actions being dispatched along with the state as follows:


Using NgRx Devtools to debug state
Banana 🍌
Add to Bucket
Bucket
Grapes 🍇
Grapes 🍇
Elements
Console
Redux
Actions
Settings
Reset
Revert
Sweep
Commit
NgRx Store Dev
filter...
State
Action
Sta
Tree
Chart
Raw

```
@ngrx/store/init                11:03:47.86
[Bucket] Add Fruit              +00:05.11
[Bucket] Add Fruit              +00:00.52
[Bucket] Add Fruit              +00:01.76
[Bucket] Remove Fruit           +00:02.63
```

```
▾ bucket (pin)
  ▾ 0 (pin)
      id (pin): 1678053833507
      name (pin): "Grapes 🍇"
  ▾ 1 (pin)
      id (pin): 1678053832979
      name (pin): "Grapes 🍇"
```

Figure 6.5: Redux DevTools showing bucket actions and state

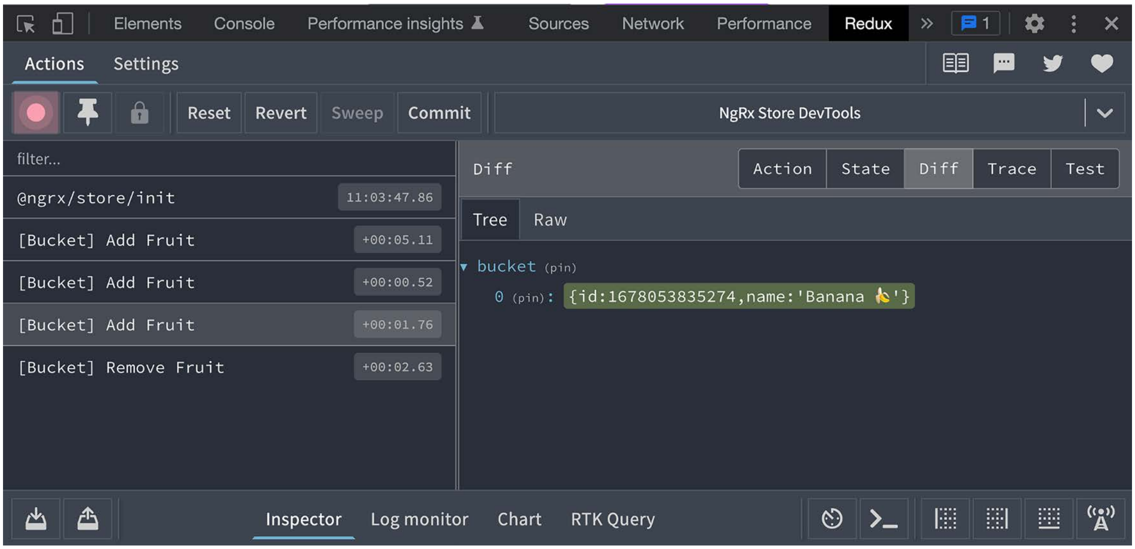
Great! You've just learned how to use the Redux DevTools extension to see your NgRx state and the actions being dispatched.

How it works...

It is important to understand that NgRx is a combination of Angular and RxJS, resulting in different patterns being implemented including Redux. By using the Store Devtools package and the Redux DevTools extension, we're able to debug the app easily, which helps us find potential bugs, predict state changes, and be more transparent about what's happening behind the scenes in the `@ngrx/store` package.

There’s more...

You can also see the difference that an action caused within an app’s state, i.e., when we add an item to the bucket and we remove an item from the bucket. See *Figure 6.6* and *Figure 6.7* for each case respectively:

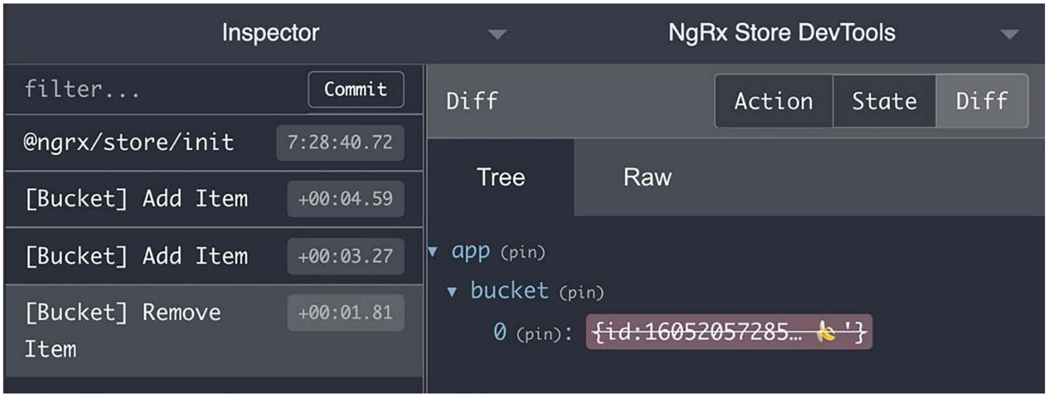

Elements
Console
Performance insights
Sources
Network
Performance
Redux
1
Actions
Settings
Reset
Revert
Sweep
Commit
NgRx Store DevTools
filter...
Diff
Action
State
Diff
Trace
Test
Tree
Raw
Inspector
Log monitor
Chart
RTK Query

```
@ngrx/store/init        11:03:47.86
[Bucket] Add Fruit      +00:05.11
[Bucket] Add Fruit      +00:00.52
[Bucket] Add Fruit      +00:01.76
[Bucket] Remove Fruit   +00:02.63
```

```
▾ bucket (pin)
  0 (pin): {id:1678053835274,name:'Banana 🍌'}
```

Figure 6.6: Add Fruit action in Redux Devtools

Notice the green background around the bucket item in *Figure 6.6*. This represents an addition to the state. You can see the `ReemoveFruit` action in the following image:


Inspector
NgRx Store DevTools
filter...
Commit
Diff
Action
State
Diff
Tree
Raw

```
@ngrx/store/init        7:28:40.72
[Bucket] Add Item       +00:04.59
[Bucket] Add Item       +00:03.27
[Bucket] Remove Item    +00:01.81
```

```
▾ app (pin)
  ▾ bucket (pin)
    0 (pin): ~~{id:16052057285… 🍌'}~~
```

Figure 6.7: Remove Fruit action in Redux Devtools

Now notice the red background and a strikethrough of the `Diff`. This represents removal from the state.

See also

- NgRx Store Devtools documentation (`https://ngrx.io/guide/store-devtools`)

Using NgRx selectors to select and render state in components

In the previous recipes, we created some actions, a single reducer, and we integrated devtools to observe the state changes. However, our bucket application still renders the data using some variables in `BucketService`. In this recipe, we're going all in with NgRx. We're going to render the bucket items from the state as we already are saving them in the NgRx store.

Getting ready

The app that we are going to work with resides in `start/apps/chapter06/ngrx-selectors` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ngrx-selectors
```

This should open the app in a new browser tab, and you should see the following:

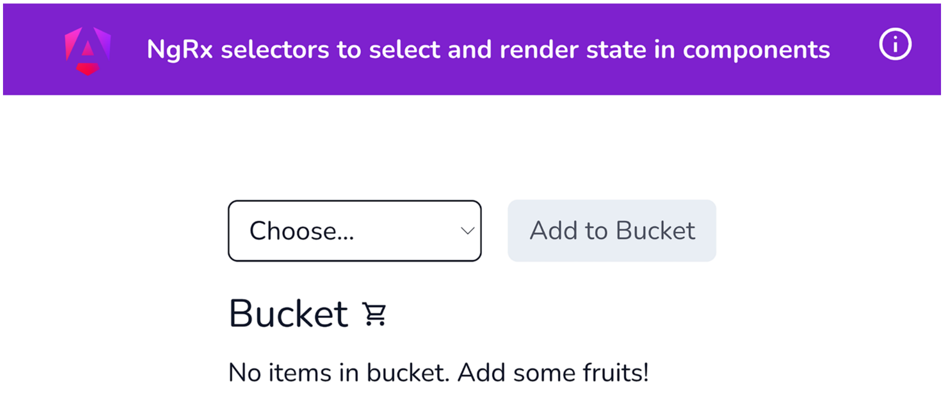

NgRx selectors to select and render state in components
Choose...
Add to Bucket
Bucket
No items in bucket. Add some fruits!

Figure 6.8: Using the ngrx-selectors app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it…

All we must do in this recipe is to work with NgRx selectors. The store, actions, and the reducer have already been set up. *Easy peasy*! Let's get started!

We'll first show the fruits in our bucket on the **Home** page and, to do that, we must create our first NgRx selector:

1. Create a new file inside the `store` folder. Name it `bucket.selectors.ts` and add the following code to it:

```
import { createFeatureSelector } from '@ngrx/store';
import { IFruit } from '../interfaces/fruit.interface';
export const selectBucket =
createFeatureSelector<ReadonlyArray<IFruit>>('bucket');
```

   Now that we have the selector in place, let's use it in the `BucketComponent` class.

2. Modify the `bucket.component.ts` file as follows to reassign the `$bucket` observable:

```
...
import { selectBucket } from '../store/bucket.selectors';
      ...
export class BucketComponent implements OnInit {
  ...
  store = inject(Store);
  $bucket: Observable<IFruit[]> = this.store.select(
    selectBucket);

  ngOnInit(): void {
    this.bucketService.loadItems();
  }
  ...
}
```

   Refresh the app and try adding and removing items from the bucket. You can see that we still get the bucket items rendered. This time, it is from the NgRx store.

3. Now we can remove the extra code that was managing the state from `BucketService`. Update the `BucketComponent` class in the `bucket.component.ts` file as follows:

```
//  remove the `implements onInit` below
export class BucketComponent implements OnInit {
  bucketService = inject(BucketService);// ← remove
  selectedFruit: Fruit = '' as Fruit;
  ...
  ngOnInit(): void {// ← remove this method
    this.bucketService.loadItems();
  }

  addSelectedFruitToBucket() {
    const newFruit: IFruit = {...};
    this.store.dispatch(...);
    this.bucketService.addItem(newFruit);// ← remove
  }
  deleteFromBucket(fruit: IFruit) {
    this.store.dispatch(...);
    this.bucketService.removeItem(fruit);// ← remove
  }
}
```

Once you've removed the code, make sure to remove unused dependencies (imports) from the file as well. Try the app and you'll see it still works. *Yay!*

4. Now we can update the `bucket.service.ts` file to not keep the bucket items in `BehaviorSubject`. Update the file as follows:

```
...
import { Injectable } from '@angular/core';
import { IFruit } from '../interfaces/fruit.interface';
import { IBucketService } from '../interfaces/bucket-service';

@Injectable({
  providedIn: 'root',
})
export class BucketService implements IBucketService {
  storeKey = 'bucket_ngrx-selectors';
  loadItems() {
    return JSON.parse(window.localStorage.getItem(
      this.storeKey) || '[]');
  }
  saveItems(items: IFruit[]) {
    window.localStorage.setItem(
      this.storeKey, JSON.stringify(items));
r}
}
```

Notice that we've removed the `addItem` and `removeItem` functions and added the `saveItems` method.

5. You will see that TypeScript is angry. That's because `BucketService` doesn't implement the `IBucketService` interface anymore. Update the `bucket-service.ts` file to update the interface as follows:

```
import { IFruit } from './fruit.interface';
export interface IBucketService {
  loadItems(): void;
  saveItems(fruit: IFruit[]): void;
}
```

And that's it! You've finished the recipe. You will notice that as soon as we refresh the app, we lose the bucket items. But fear not – we'll bring them back in the next recipe.

How it works...

In this recipe, we already had actions and selectors set up. However, we were only dispatching actions to the store to add/remove items. The components were still using the `BucketService` and the Observable from it to manage the state and to render the items. Since we only wanted to work with NgRx for the entire process, we introduced a selector in our application to select the bucket items.

Notice that we use the `createFeatureSelector` method from `@ngrx/store` package. This allows us to select a feature from the application. For example, your application can have features like profile, users, events, and settings. And ideally, we create feature selectors to extract only the target feature's data instead of the whole NgRx state object.

Then we replace the `$bucket` property in our `BucketComponent` class using the selector `selectBucket()` we just created. Notice that the selector returns a `Readonly` array. This ensures we are not mutating the actual data and that is is read-only. Finally, we remove the unnecessary functions from the `BucketService` class and remove the usage of those functions from the `BucketComponent` class as well.

If you look at the app holistically now, we can add and remove items to the bucket using NgRx and can use selectors to render the items in the UI. However, as soon as you refresh the app, it all goes away since the data is not persistent; even though we have the functions to get and save items using `LocalStorage`. The ideal way of handling this would be to use NgRx effects. However, you are going to learn them in the next recipe. You can challenge yourself to come back to this recipe again to implement an effect that saves the updated bucket list to storage.

See also

- NgRx selectors documentation (`https://ngrx.io/guide/store/selectors`)

Using NgRx effects to fetch data from API calls

In this recipe, you'll learn how to use NgRx effects using the `@ngrx/effects` package. We have an app that already has `@ngrx/store` and `@ngrx/store-devtools` installed. And we are able to add and remove items from the bucket. However, in this recipe, we'll use a server to receive, store, add, and remove items from a bucket, i.e., the data will live in the NgRx store as well as on the backend.

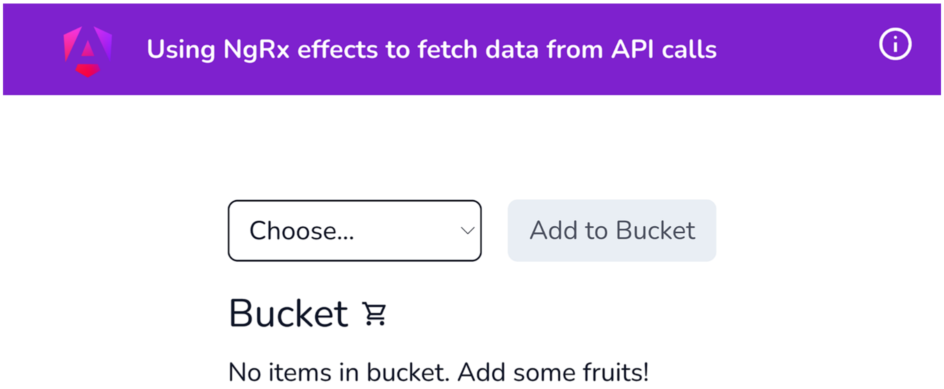
Getting ready

The app that we are going to work with resides in `start/apps/chapter06/ngrx-effects` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project with the backend app:

```
npm run serve ngrx-effects with-server
```

This should open the app in a new browser tab, and you should see the following:



Using NgRx effects to fetch data from API calls
Choose...
Add to Bucket
Bucket
No items in bucket. Add some fruits!

Figure 6.9: ngrx-effects app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

We have an Angular application having a single page containing a bucket. You can add fruits to your buckets and remove items from the bucket. We already have the `@ngrx/store`, `@ngrx/store-devtools`, and `@ngrx/effects` packages installed in the workspace, so you don't have to install them. However, when you work on a standalone app (or on some of your own projects), you could add NgRx effects by running the following command:

```
npm install --save @ngrx/effects
```

Now let's follow these steps:

1. We'll update the `bucket.actions.ts` file to add some actions including the API call. Update the file as follows:

```
import { createActionGroup, props, emptyProps } from '@ngrx/store';
import { IFruit } from '../interfaces/fruit.interface';

export const BucketActions = createActionGroup({
  source: 'Bucket',
  events: {
    'Get Bucket': emptyProps(),
    'Get Bucket Success': props<{ bucket: IFruit[] }>(),
    'Get Bucket Failure': props<{ error: string }>(),
    'Add Fruit': props<{ fruit: IFruit }>(),
    'Add Fruit Success': props<{ fruit: IFruit }>(),
    'Add Fruit Failure': props<{ error: string }>(),
    'Remove Fruit': props<{ fruitId: number }>(),
      'Remove Fruit Success': props<{ fruitId: number }>(),
      'Remove Fruit Failure': props<{ error: string }>(),
  },
});
```

2. Create a file in the `store` folder named `bucket.effects.ts` and add the following code to it:

```
import { Injectable } from '@angular/core';
import { Actions, ofType, createEffect } from '@ngrx/effects';
import { of } from 'rxjs';
import { catchError, exhaustMap, map } from 'rxjs/operators';
import { BucketService } from '../bucket/bucket.service';
import { BucketActions } from './bucket.actions';

@Injectable()
export class BucketEffects {
  getBucket$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BucketActions.getBucket),
      exhaustMap(() =>
        this.bucketService.getBucket().pipe(
          map(({ bucket }) => BucketActions
            .getBucketSuccess({ bucket })),
          catchError((error) => of(BucketActions
            .getBucketFailure({ error })))
        )
      )
    )
  );
  constructor(
    private actions$: Actions,
    private bucketService: BucketService
  ) {}
}
```

3. Now we can provide the effects with the `BucketEffects` class in the `app.config.ts` file as follows:

```
...
import { provideEffects } from '@ngrx/effects';
...
import { BucketEffects } from './app/store/bucket.effects';
...
```

```
bootstrapApplication(AppComponent, {
  providers: [
    ...,
    provideHttpClient(),
    provideEffects([BucketEffects]),
  ],
}).catch((err) => console.error(err));
```

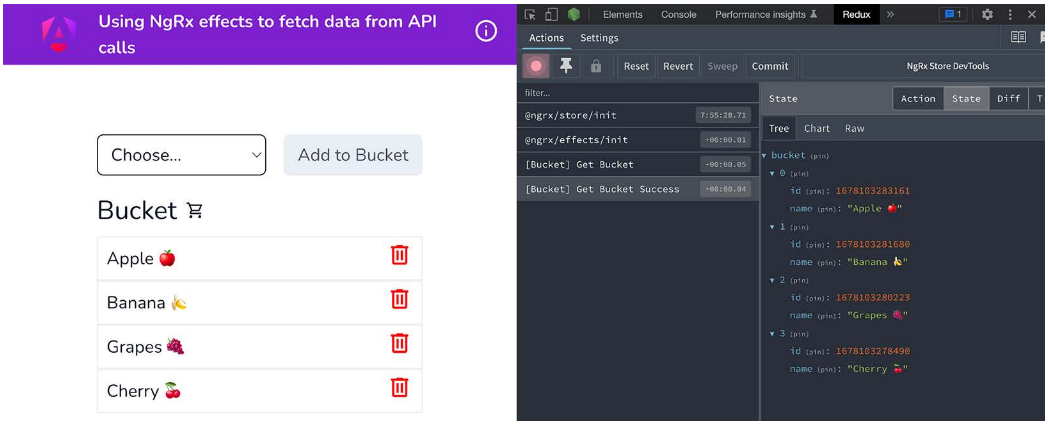
4. We'll now update the `bucket.reducer.ts` file to add or remove items from the state on the success events of the HTTP calls. Also, to set the bucket items retrieved from the server on app start. Update the file as follows:

```
...
export const initialState: ReadonlyArray<IFruit> = [];
export const bucketReducer = createReducer(
  initialState,
  on(BucketActions.getBucketSuccess,
    (_state, { bucket }) => {
    return bucket;
  }),
  on(BucketActions.addFruitSuccess, (_state, { fruit }) => {
    console.log({ fruit });
    return [fruit, ..._state];
  }),
  on(BucketActions.removeFruitSuccess, (_state, { fruitId }) => {
    console.log({ fruitId });
    return _state.filter((fr) => fr.id !== fruitId);
  })
);
```

5. We'll now dispatch the `getBucket` action from `bucket.componen.ts` when the component is mounted. Update `bucket.component.ts` as follows:

```
import { CommonModule } from '@angular/common';
import { Component, inject, OnInit } from '@angular/core';
...
@Component({...})
export class BucketComponent implements OnInit {
  ...
  ngOnInit() {
    this.store.dispatch(BucketActions.getBucket());
  }
  ...
}
```

Now that we have linked everything, refresh the app and you'll see the bucket items from the server in the app. You can see the actions in Redux Devtools as follows:

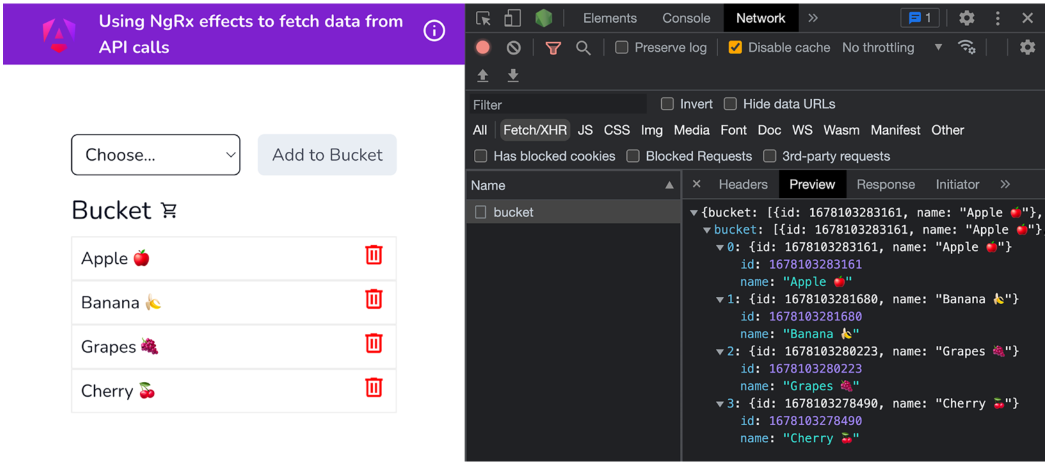

Using NgRx effects to fetch data from API calls
Choose...
Add to Bucket
Bucket
Apple 🍎
Banana 🍌
Grapes 🍇
Cherry 🍒
Elements
Console
Performance insights
Redux
Actions
Settings
Reset
Revert
Sweep
Commit
NgRx Store DevTools
filter...
State
Action
State
Diff
Tree
Chart
Raw

```
@ngrx/store/init                 7:55:28.71
@ngrx/effects/init               +00:00.01
[Bucket] Get Bucket              +00:00.05
[Bucket] Get Bucket Success      +00:00.04
```

```
▾ bucket (pin)
  ▾ 0 (pin)
      id (pin): 1678103283161
      name (pin): "Apple 🍎"
  ▾ 1 (pin)
      id (pin): 1678103281680
      name (pin): "Banana 🍌"
  ▾ 2 (pin)
      id (pin): 1678103280223
      name (pin): "Grapes 🍇"
  ▾ 3 (pin)
      id (pin): 1678103278490
      name (pin): "Cherry 🍒"
```

Figure 6.10: Getting the bucket from the server using NgRx effects

You should also be able to see the network call (see *Figure 6.10*) that occurred due to the `[Bucket] Get Bucket` action in *Figure 6.9*.


Using NgRx effects to fetch data from API calls
Choose...
Add to Bucket
Bucket
Apple 🍎
Banana 🍌
Grapes 🍇
Cherry 🍒
Elements
Console
Network
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
Doc
WS
Wasm
Manifest
Other
Has blocked cookies
Blocked Requests
3rd-party requests
Name
bucket
Headers
Preview
Response
Initiator

```
▾{bucket: [{id: 1678103283161, name: "Apple 🍎"},
  ▾bucket: [{id: 1678103283161, name: "Apple 🍎"}
    ▾0: {id: 1678103283161, name: "Apple 🍎"}
       id: 1678103283161
       name: "Apple 🍎"
    ▾1: {id: 1678103281680, name: "Banana 🍌"}
       id: 1678103281680
       name: "Banana 🍌"
    ▾2: {id: 1678103280223, name: "Grapes 🍇"}
       id: 1678103280223
       name: "Grapes 🍇"
    ▾3: {id: 1678103278490, name: "Cherry 🍒"}
       id: 1678103278490
       name: "Cherry 🍒"
```

Figure 6.11: @ngrx/effects initiating network calls

If you try to add or remove items, you'll see that they don't work. That's because we changed our reducer to act when there's an `addFruitSuccess` or `removeFruitSuccess` event in *step 4*.

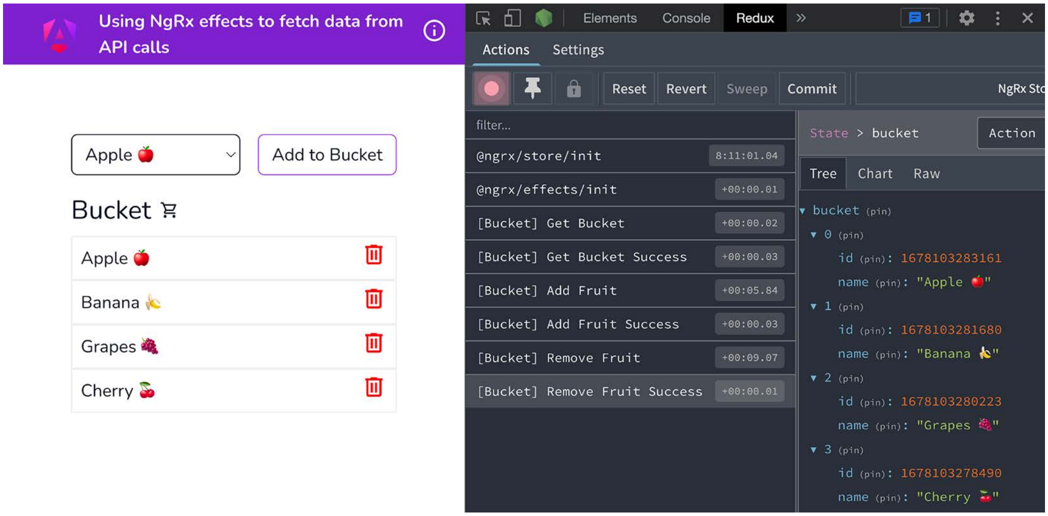
6. Now we will add the effect for `addFruit` in `bucket.effects.ts`. Update the file as follows:

```
...
export class BucketEffects {
  ...
  addItem$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BucketActions.addFruit),
      exhaustMap((action) =>
        this.bucketService.addItem(action.fruit).pipe(
          map(({ fruit }) => BucketActions
            .addFruitSuccess({ fruit })),
          catchError((error) => of(
            BucketActions.addFruitFailure({ error })))
        )
      )
    )
  );
  ...
}
```

7. Let's add the effect for `removeFruit` as well, in `bucket.effects.ts`. Update the file as follows:

```
...
export class BucketEffects {
  ...
  removeItem$ = createEffect(() =>
    this.actions$.pipe(
      ofType(BucketActions.removeFruit),
      exhaustMap((action) =>
        this.bucketService.removeItem(
          action.fruitId).pipe(
          map(() =>
            BucketActions.removeFruitSuccess({ fruitId:
              action.fruitId })
          ),
          catchError((error) => of(
            BucketActions.removeFruitFailure({ error })))
        )
      )
    )
  );
  ...
}
```

If you refresh the app now, add an apple, and remove it, you should see the state in Redux Devtools, as shown in the following figure:



```
@ngrx/store/init                8:11:01.04
@ngrx/effects/init              +00:00.01
[Bucket] Get Bucket             +00:00.02
[Bucket] Get Bucket Success     +00:00.03
[Bucket] Add Fruit              +00:05.84
[Bucket] Add Fruit Success      +00:00.03
[Bucket] Remove Fruit           +00:09.07
[Bucket] Remove Fruit Success   +00:00.01
```

```
▾ bucket (pin)
  ▾ 0 (pin)
      id (pin): 1678103283161
      name (pin): "Apple 🍎"
  ▾ 1 (pin)
      id (pin): 1678103281680
      name (pin): "Banana 🍌"
  ▾ 2 (pin)
      id (pin): 1678103280223
      name (pin): "Grapes 🍇"
  ▾ 3 (pin)
      id (pin): 1678103278490
      name (pin): "Cherry 🍒"
```

Figure 6.12: NgRx effects getting a bucket and adding and removing items

Notice in *Figure 6.11* that the state shows 4 items, shown in the UI. Also, notice the sequence of the actions dispatched. Every API/HTTP action results in the following `Success` action in this case.

Great! You now know how to use NgRx effects in your Angular apps. See the next section to understand how NgRx effects work.

How it works...

In order for the NgRx effects to work, we needed to install the `@ngrx/effects` package, create an effect, and register it in an array of effects (root effects) in the `main.ts` file. An NgRx effect is essentially something that listens to an action, does something (we call this something a side-effect), and then returns an action as a result. When an action is dispatched to the store from any component or even from another effect, the registered effect does the job you want it to do and is supposed to dispatch another action in return. For HTTP calls, we usually have three actions—that is, the main action, and the following success and failure actions. Ideally, on the success action (and perhaps on the failure action too), you would want to update some of your state variables. In *step 4*, inside the `BucketEffects` class, you'll notice that we inject the `Action` service from NgRx and `BucketService`. Then we use the `createEffect` function to create the effect. The effect itself listens to the `Action` stream for a particular event using the `ofType` operator. We then use the `exhaustMap` operator to perform the HTTP call, which returns an observable. Finally, we use the `map` operator to return the success action and the `catchError` operator to return the failure action.

See also

- NgRx effects documentation (`https://ngrx.io/guide/effects`)

Using NgRx Component Store to manage the state of a component

In this recipe, you'll learn how to use the **NgRx Component Store** and how to use it instead of the push-based `Subject/BehaviorSubject` pattern with services for maintaining a component's state. We'll also see how this can facilitate cross-component communication using the **Component Store**.

Remember that `@ngrx/component-store` is a standalone library and doesn't correlate with Redux or `@ngrx/store`, and so on.

Getting ready

The app that we are going to work with resides in `start/apps/chapter06/ngrx-component-store` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project with the backend app:

   ```
   npm run serve ngrx-component-store
   ```

   This should open the app in a new browser tab, and you should see the following:

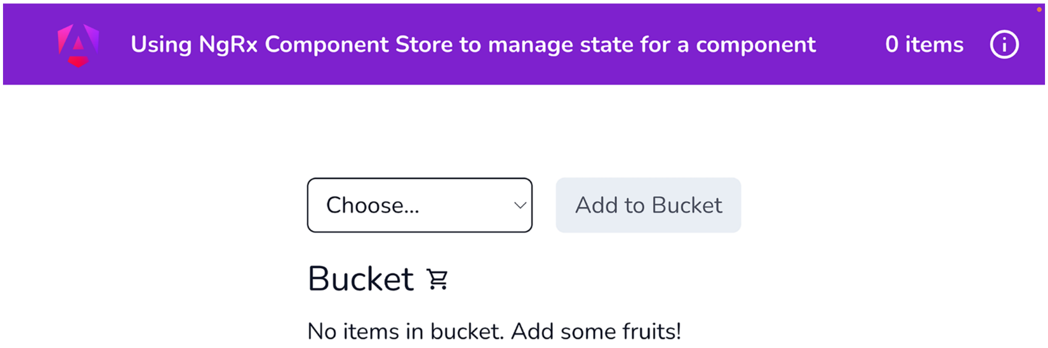

Using NgRx Component Store to manage state for a component
0 items
Choose...
Add to Bucket
Bucket
No items in bucket. Add some fruits!

Figure 6.13: ngrx-component-store app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it…

We have our favorite bucket app that we've used in a lot of recipes so far. The state of the bucket right now is stored in the component itself. This limits the state to being just in the component and we can't use the bucket's length in the top header. One way could be using `BehaviorSubject` for this. But we'd rather use NgRx Component Store. We already have `@ngrx/component-store` package installed in the monorepository. But if you're working on a new project, you would install the package as follows:

```
npm install @ngrx/component-store
```

We're going to build our component store in `BucketService`.

1. Let's make it compatible with `ComponentStore`. In order to do that, we'll create an interface for the bucket state, extend `BucketService` from `ComponentStore`, and initialize the service by calling the `super` method. Update the `bucket.service.ts` file as follows:

   ```
   ...
   import { ComponentStore } from '@ngrx/component-store';
   import { IFruit } from '../interfaces/fruit.interface';

   export interface BucketState {
     bucket: IFruit[];
   }
   ...
   export class BucketService extends ComponentStore<
     BucketState> {
     storeKey = 'bucket_ngrx-component-store';

     constructor() {
       super({ bucket: [] });
     }

     loadItems() {...}
     saveItems(items: IFruit[]) {...}
   }
   ```

   None of this will make sense until we actually show the data from `ComponentStore`. Let's work on that now.

2. Let's create a `bucket$` observable in `BucketService` as follows:

   ```
   ...
   import { Observable } from 'rxjs/internal/Observable';
   import { IFruit } from '../interfaces/fruit.interface';
   ...
   export class BucketService extends ComponentStore<BucketState> {
     storeKey = 'bucket_ngrx-component-store';
     readonly bucket$: Observable<IFruit[]> = this.select(
       (state) => state.bucket);
     ...
   }
   ```

3. First, let's make sure that we can initialize the component store using the values from `localStorage`. We'll update the `constructor` method in the `bucket.service.ts` file as follows to use the `loadItems` method:

```
  constructor() {
    super({ bucket: [] });
    this.setState({
      bucket: this.loadItems(),
    });
  }
```

4. Now we'll add the methods for adding and removing fruit items to and from the bucket:

```
export class BucketService extends ComponentStore<BucketState> {
  ...

  readonly addItem = this.updater((state: BucketState,
    fruit: IFruit) => {
    const bucketUpdated = [fruit, ...state.bucket];
    this.saveItems(bucketUpdated);
    return {
      bucket: bucketUpdated,
    };
  });
  readonly removeItem = this.updater((state: BucketState,
    fruitId: number) => {
    const bucketUpdated = state.bucket.filter((fr) =>
      fr.id !== fruitId);
    this.saveItems(bucketUpdated);
    return {
      bucket: bucketUpdated,
    };
  });
  ...
}
```

5. Now that we have the Component Store set up, let's use it in the `bucket.component.ts` file as follows:

```
...
import { BucketService } from './bucket.service';

@Component({...})
export class BucketComponent {
  ...
  bucket: IFruit[] = []; //← remove
  store = inject(BucketService); //← add
  bucket$ = this.store.bucket$; //← add

  addSelectedFruitToBucket() {
    const newFruit: IFruit = {
      id: Date.now(),
      name: this.selectedFruit,
    };
    this.store.addItem(newFruit);
  }

  deleteFromBucket(fruit: IFruit) {
    this.store.removeItem(fruit.id);
  }
}
```

6. We can now use the `bucket$` observable in the template. Update `bucket.component.html` as follows:

```
  <div class="fruits" *ngIf="bucket$ | async as bucket"
    [@listItemAnimation]="bucket.length">
    <ng-container *ngIf="bucket.length > 0; else
      bucketEmptyMessage">
    ...
    </ng-container>
    <ng-template #bucketEmptyMessage>...</ng-template>
  </div>
```

   With this change, you should see the app working. If you now try to add or remove items, you should be able to see them being reflected in the app. And this is all being done using Component Store now.

7. The next thing we need to do is to use the bucket's length in the application's header. Let's create a new state selector in the `bucket.service.ts` file as follows:

```
...
export class BucketService extends ComponentStore<BucketState> {
  storeKey = 'bucket_ngrx-component-store';
  readonly bucket$: Observable<IFruit[]> =
    this.select((state) => state.bucket);
  readonly bucketLength$: Observable<number> =
    this.select(
      (state) => state.bucket.length
    );
    ...
  }
```

8. And we can now use the component store in the app's header. First, let's import the store in `app.component.ts` as follows:

```
...
import { BucketService } from './bucket/bucket.service';
...
export class AppComponent {
  store = inject(BucketService);
  bucketLength$ = this.store.bucketLength$;
}
```

9. We can now use the `bucketLength$` observable in `app.component.html` as follows:

```
<span class="mr-4">{{bucketLength$ | async}} items</span>
```

And voila! If you add and remove items from the bucket, you should be able to see the bucket's length in the header as well:

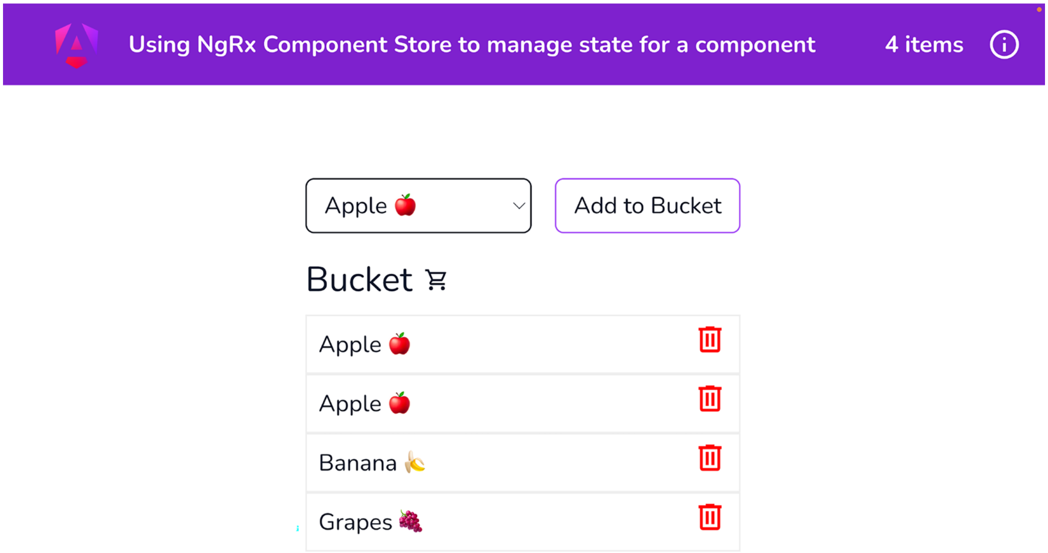

Using NgRx Component Store to manage state for a component
4 items
Apple 🍎
Add to Bucket
Bucket 🛒
Apple 🍎
Apple 🍎
Banana 🍌
Grapes 🍇

Figure 6.14: Bucket length in header via Component Store

Congratulations! You finished the recipe. See the next section to understand how it works.

How it works...

As mentioned earlier, `@ngrx/component-store` is a standalone package that can easily be installed in your Angular apps without having to use `@ngrx/store`, `@ngrx/effects` and so on. It is supposed to replace the usage of `BehaviorSubject` in Angular services, and that's what we did in this recipe. We covered how to initialize `ComponentStore` and how to set the initial state using the `setState` method when we already had the values without accessing the state, and we learned how to create updater methods that can be used to update the state, as they can access the state and allow us to even pass arguments for our own use cases.

See also

- `@ngrx/component-store` documentation (`https://ngrx.io/guide/component-store`)
- Effects in `@ngrx/component-store` documentation (`https://ngrx.io/guide/component-store/effect`)

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

7

Understanding Angular Navigation and Routing

One of the most amazing things about Angular is that it is an entire ecosystem (a framework) rather than a library. In this ecosystem, the Angular router is one of the most critical blocks to learn about and understand. In this chapter, you'll learn some really cool techniques for routing and navigation in Angular. You'll learn about how to guard your routes, listen to route changes, and configure global actions on route changes.

The following are the recipes we're going to cover in this chapter:

- Creating routes in an Angular (standalone) app
- Lazily loaded routes in Angular
- Preloading route strategies
- Authorized access to routes using route guards
- Working with route parameters
- Showing a global loader between route changes

Technical requirements

For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter07`.

Creating routes in an Angular (standalone) app

If you ask me about how we used to create projects for web applications 7–8 years ago, you'll be astonished to learn how difficult it was. Luckily, the tools and standards have evolved in the software development industry and, when it comes to Angular, starting a project is super easy. With the Angular standalone apps, the configuration is much smaller for the application's bootstrap process and for routing. In this recipe, you're going to implement some routes in a fresh Angular application.

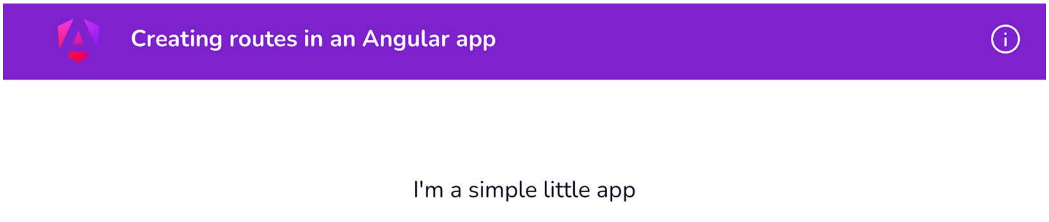
Getting ready

The app that we are going to work with resides in `start/apps/chapter07/ng-basic-routing` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-basic-routing
```

This should open the app in a new browser tab. You should see the following:



Creating routes in an Angular app

I'm a simple little app

Figure 7.1: ng-basic-routes app running on http://localhost:4200

How to do it...

We'll configure routing in the app, add some routes, and add the links to those routes in the headers. Let's get started:

1. First, create a file named `app.routes.ts` inside the app folder and add the following code to it:

```
import { Route } from '@angular/router';
export const appRoutes: Route[] = [];
```

2. Now update the file `app.config.ts` in the `src` folder with the following changes:

```
import { ApplicationConfig } from '@angular/core';
import {
  provideRouter,
  withEnabledBlockingInitialNavigation,
} from '@angular/router';
import { appRoutes } from './app.routes';
export const appConfig: ApplicationConfig = {
  providers: [provideRouter(appRoutes,
withEnabledBlockingInitialNavigation())],
};
```

3. Now let's create some components (as pages). We're going to create a home page and an `about` page. Running the following `npx` command in the terminal from the `start` folder in the workspace. If asked, you can use the `@nx/angular:component` schematics and choose the "As provided" action:

   ```
   nx g c home --directory apps/chapter07/ng-basic-routing/src/app/home
   --standalone
   nx g c about --directory apps/chapter07/ng-basic-routing/src/app/about
   --standalone
   ```

   You should see two new folders created in the `app` folder of the project now.

4. We will configure the routes now. Update the `app.routes.ts` file that we created in *step 1* as follows:

   ```
   import { Route } from '@angular/router';
   import { AboutComponent } from './about/about.component';
   import { HomeComponent } from './home/home.component';
   export const appRoutes: Route[] = [{
     path: '',
     pathMatch: 'full',
     redirectTo: 'home'
   }, {
     path: 'home',
     component: HomeComponent
   }, {
     path: 'about',
     component: AboutComponent
   }
   ];
   ```

5. All that remains is to hook the routes in the view using `<router-outlet>` in the template. Update the `app.component.html` file as follows:

   ```
   <!-- Toolbar -->
   ...
   <main class="content" role="main">
     <router-outlet></router-outlet>
   </main>
   ```

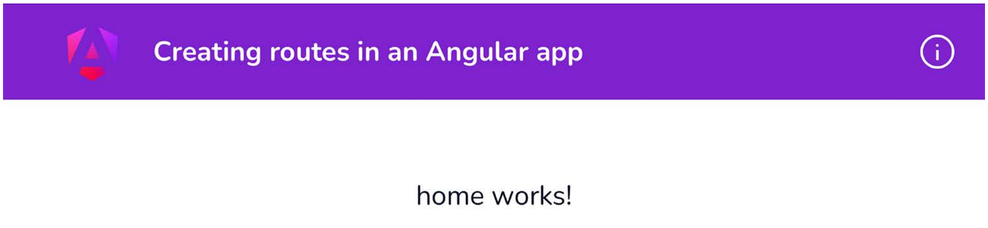
   But wait! That crashes the app. That's because the `AppComponent` doesn't understand routing yet.

6. Update the `app.component.ts` file to add the `RouterModule` in it as follows:

   ```
   ...
   import { RouterModule } from '@angular/router';
   @Component({
     ...,
   ```

```
  imports: [CommonModule, RouterModule],
})
export class AppComponent {}
```

And voilà ! You should be able to see the `home` component now, as follows:



Creating routes in an Angular app

home works!

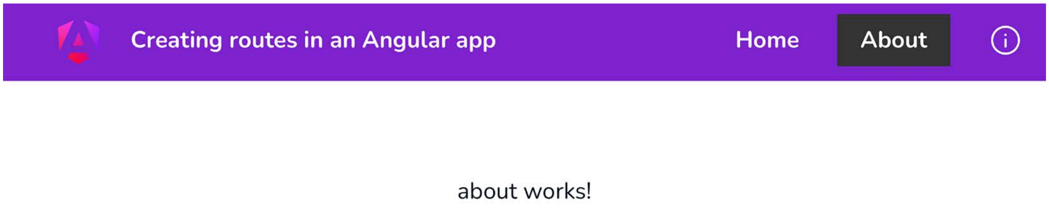
Figure 7.2: The home component in the ng-basic-routes app

If you look at the URL, it defaults to `http://localhost:4200/home` even if you try to go to `https://localhost:4200`.

7. Finally, we'll add the links for the routes in the header. Update `app.component.html` as follows:

```
<!-- Toolbar -->
<div class="toolbar" role="banner">
  ...
  <div class="spacer"></div>
  <div class="route-links">
    <div
      class="route-links__route-link"
      routerLink="/home"
      routerLinkActive="route-links__route-link--active"
    >
      Home
    </div>
    <div
      class="route-links__route-link"
      routerLink="/about"
      routerLinkActive="route-links__route-link--active"
    >
      About
    </div>
  </div>
  ...

<main class="content" role="main">...</main>
```

Now, if you click the **About** link at the top, you should be able to see the `about` route, as follows:



about works!

Figure 7.3: The about component in the ng-basic-routes app

Awesome! Within a few minutes, and with the help of Angular router and NX CLI, we were able to create a home page and an `about` page, and we configured routing as well. The wonders of the modern web!

Now that you know how basic routing is implemented, the next section will help you to understand how it works.

How it works...

When we create an Angular app without the `--routing` flag, we don't get the `app.routes.ts` file or have the configuration added in the `main.ts` file. For this example, we worked with an app that already had this flag set. If you were to start a new project, you might as well just use the `--routing` flag while creating the app to have routing set up in the app from the start. Then we created some standalone components using the `nx g c <component name> -standalone` command. If you were using the Angular CLI instead of an NX monorepo, you would just replace `nx` with `ng` in the preceding command, and the rest would be the same. Using `npx` allows us to use the `nx` package without globally installing it in our system. Since we have a standalone app, i.e., the `AppComponent` is a standalone component (notice the `standalone: true` in the `app.component.ts` file), we had to import the `RouterModule` to it to be able to use the `<router-outlet>` and the `routeLink` attribute on the links. We configured the routes by using the `provideRouter` function in the `main.ts` file with the routes since we don't have an `AppModule` (because the app is a standalone app).

See also

- Angular router docs: `https://angular.io/guide/router`
- Angular standalone components: `https://angular.io/guide/standalone-components`

Lazily loaded routes in Angular

In the previous recipe, we learned how to create a basic routing app with eagerly loaded routes. In this recipe, you'll learn how to work with feature modules to lazily load them instead of loading them when the app loads. For this recipe, we'll assume that we already have the routes in place, and we just need to load them lazily.

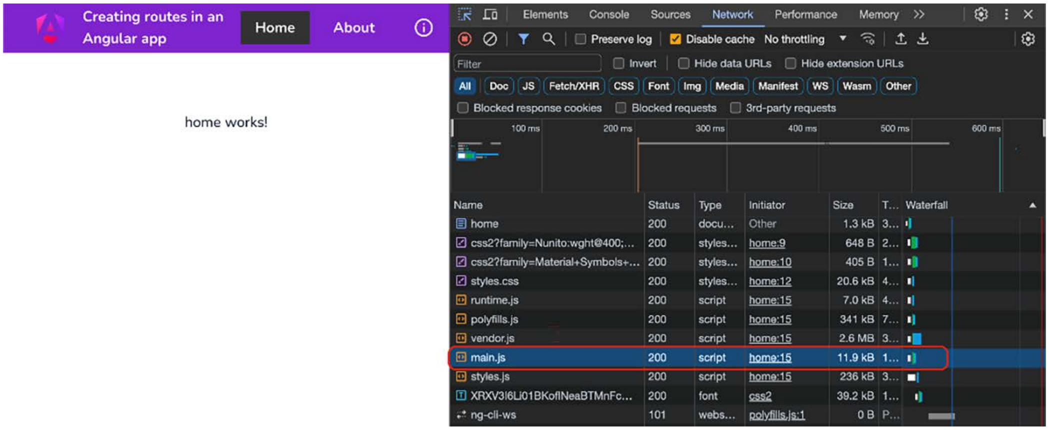
Getting ready

The app that we are going to work with resides in `start/apps/chapter07/ng-lazy-routing` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-lazy-routing
```

This should open the app in a new browser tab. If you open the **Network** tab, you should see the following:


Creating routes in an Angular app
Home
About
Elements
Console
Sources
Network
Performance
Memory
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
Hide extension URLs
All
Doc
JS
Fetch/XHR
CSS
Font
Img
Media
Manifest
WS
Wasm
Other
Blocked response cookies
Blocked requests
3rd-party requests
100 ms
200 ms
300 ms
400 ms
500 ms
600 ms

home works!

Name	Status	Type	Initiator	Size	T...	Waterfall
home	200	docu...	Other	1.3 kB	3...	
css2?family=Nunito:wght@400;...	200	styles...	home:9	648 B	2...	
css2?family=Material+Symbols+...	200	styles...	home:10	405 B	1...	
styles.css	200	styles...	home:12	20.6 kB	4...	
runtime.js	200	script	home:15	7.0 kB	4...	
polyfills.js	200	script	home:15	341 kB	7...	
vendor.js	200	script	home:15	2.6 MB	3...	
main.js	200	script	home:15	11.9 kB	1...	
styles.js	200	script	home:15	236 kB	3...	
XRXV3I6Li01BKofINeaBTMnFc...	200	font	css2	39.2 kB	1...	
ng-cli-ws	101	webs...	polyfills.js:1	0 B	P...	

Figure 7.4: ng-lazy-routing app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

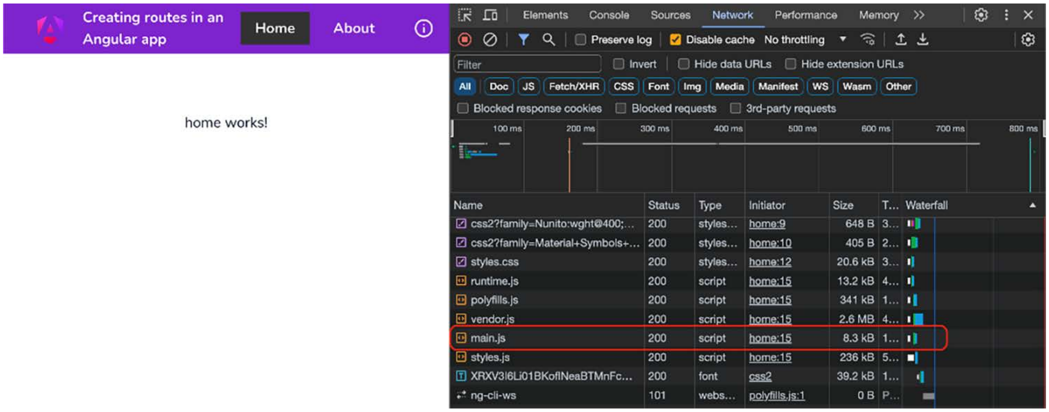
How to do it…

As shown in *Figure 7.4*, we have all the components and modules inside the `main.js` file. Therefore, the `main.js` file is about 11.9 KB in size (this could change depending on how Angular further optimizes the framework). We'll modify the code and the routing structure to achieve lazy loading. As a result, the files of the routes will load when we navigate to them. Before the *standalone components* era, this step was difficult. And even today, if you work with Angular applications using `NgModules`, you will find that it takes a lot more code to set up your routing. But since our application has only standalone components, see how easy it becomes to change it:

1. Update the `app.routes.ts` to use the `loadComponent` method instead of the `component` property in the routes, as follows:

```
import { Route } from '@angular/router';

export const appRoutes: Route[] = [{
  path: '',
  pathMatch: 'full',
  redirectTo: 'home'
}, {
  path: 'home',
  loadComponent: () => import('./home/home.component').then(m =>
m.HomeComponent)
}, {
  path: 'about',
  loadComponent: () => import('./about/about.component').then(m =>
m.AboutComponent)
}];
```

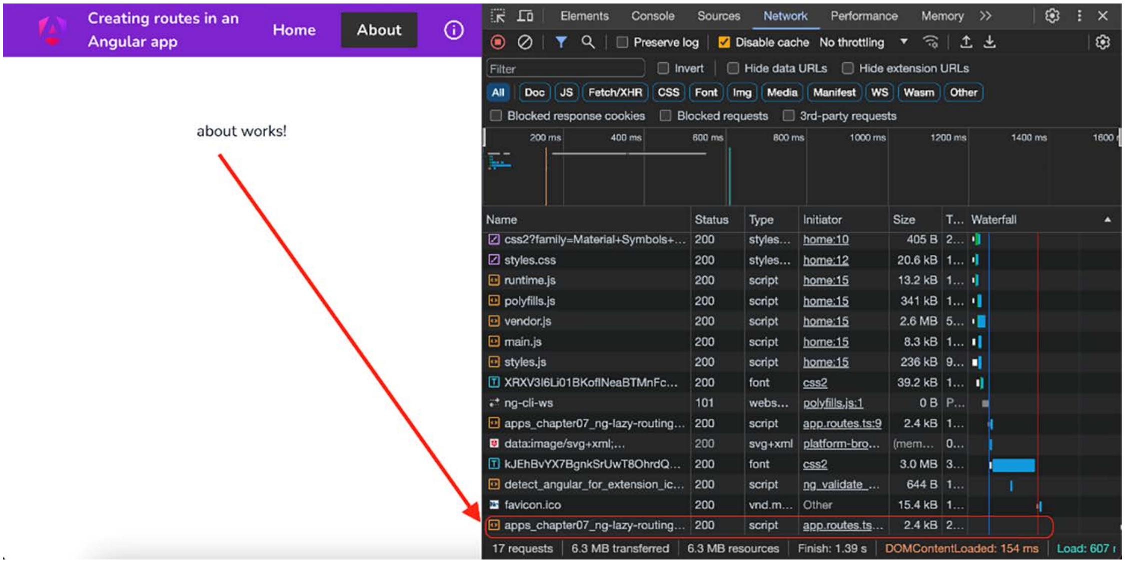
Refresh the app and you'll see that the bundle size for the `main.js` file is down to 8.3 KB, which was about 11.9 KB before. See the following screenshot:


Creating routes in an Angular app
Home
About
home works!
Elements
Console
Sources
Network
Performance
Memory
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
Hide extension URLs
All
Doc
JS
Fetch/XHR
CSS
Font
Img
Media
Manifest
WS
Wasm
Other
Blocked response cookies
Blocked requests
3rd-party requests
100 ms
200 ms
300 ms
400 ms
500 ms
600 ms
700 ms
800 ms

Name	Status	Type	Initiator	Size	T...	Waterfall
css2?family=Nunito:wght@400;...	200	styles...	home:9	648 B	3...	
css2?family=Material+Symbols+...	200	styles...	home:10	405 B	2...	
styles.css	200	styles...	home:12	20.6 kB	3...	
runtime.js	200	script	home:15	13.2 kB	4...	
polyfills.js	200	script	home:15	341 kB	1...	
vendor.js	200	script	home:15	2.6 MB	4...	
main.js	200	script	home:15	8.3 kB	1...	
styles.js	200	script	home:15	236 kB	5...	
XRXV3I6Li01BKofINeaBTMnFc...	200	font	css2	39.2 kB	1...	
ng-cli-ws	101	webs...	polyfills.js:1	0 B	P...	

Figure 7.5: Reduced size of main.js on app load

But what about the `Home` and `About` routes? And what about lazy loading? Well, tap the **About** route from the header and you'll see a new JavaScript file being downloaded in the **Network** tab specifically for the route. That's lazy loading in action! See the following screenshot:



☐ Preserve log ☑ Disable cache No throttling

☐ Invert ☐ Hide data URLs ☐ Hide extension URLs

☐ Blocked response cookies ☐ Blocked requests ☐ 3rd-party requests

Name	Status	Type	Initiator	Size	T...	Waterfall
css2?family=Material+Symbols+...	200	styles...	home:10	405 B	2...	
styles.css	200	styles...	home:12	20.6 kB	1...	
runtime.js	200	script	home:15	13.2 kB	1...	
polyfills.js	200	script	home:15	341 kB	1...	
vendor.js	200	script	home:15	2.6 MB	5...	
main.js	200	script	home:15	8.3 kB	1...	
styles.js	200	script	home:15	236 kB	9...	
XRXV3I6Li01BKofINeaBTMnFc...	200	font	css2	39.2 kB	1...	
ng-cli-ws	101	webs...	polyfills.js:1	0 B	P...	
apps_chapter07_ng-lazy-routing...	200	script	app.routes.ts:9	2.4 kB	1...	
data:image/svg+xml;...	200	svg+xml	platform-bro...	(mem...	0...	
kJEhBvYX7BgnkSrUwT8OhrdQ...	200	font	css2	3.0 MB	3...	
detect_angular_for_extension_ic...	200	script	ng_validate_...	644 B	1...	
favicon.ico	200	vnd.m...	Other	15.4 kB	1...	
apps_chapter07_ng-lazy-routing...	200	script	app.routes.ts...	2.4 kB	2...	

17 requests | 6.3 MB transferred | 6.3 MB resources | Finish: 1.39 s | DOMContentLoaded: 154 ms | Load: 607

Figure 7.6: about route being lazily loaded

Awesomesauce! You just became lazy! Just kidding. You just learned the art of lazily loading routes and feature components in your Angular app. You can now show this off to your friends as well.

How it works...

Angular works with modules and components and, usually, the features are broken down into either `NgModules`, or with standalone components, the components themselves are lazily loaded. We're going to look into the standalone components perspective. As we know, `AppComponent` serves as the entry point for the standalone Angular app and Angular will import and bundle anything that is imported in `AppComponent` during the build process, resulting in the `main.js` file. However, if we want to lazily load our routes/feature components, we need to avoid importing feature components in `AppComponent` directly or even importing them in the routes. Instead, we can use the `loadChildren` method for lazily loading other modules, or the `loadComponent` method to lazily load other standalone components. That's what we did in this recipe. It is important to note that the routes stayed the same in the `app.routes.ts` file.

See also

- Lazily loading modules in Angular: `https://angular.io/guide/lazy-loading-ngmodules`
- Lazily loading standalone components: `https://angular.io/guide/standalone-components#routing-and-lazy-loading`

Preloading route strategies

We're already familiar with how to lazily load different feature modules during navigation. Sometimes, though, you might want to preload subsequent routes to make the next route navigation instantaneous or might even want to use a custom preloading strategy based on your application's business logic. In this recipe, you'll learn about the `PreloadAllModules` strategy and also implement a custom strategy to cherry-pick which modules should be preloaded.

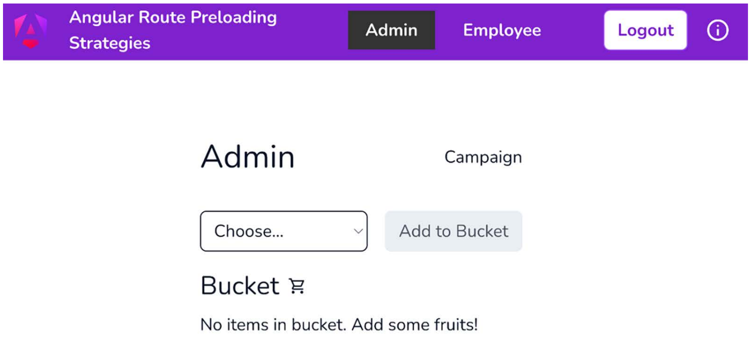
Getting ready

The app that we are going to work with resides in `start/apps/chapter07/ng-route-preload-strat` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve ng-route-preload-strat
   ```

   This should open the app in a new browser tab. If you log in as an admin, you should see the following:

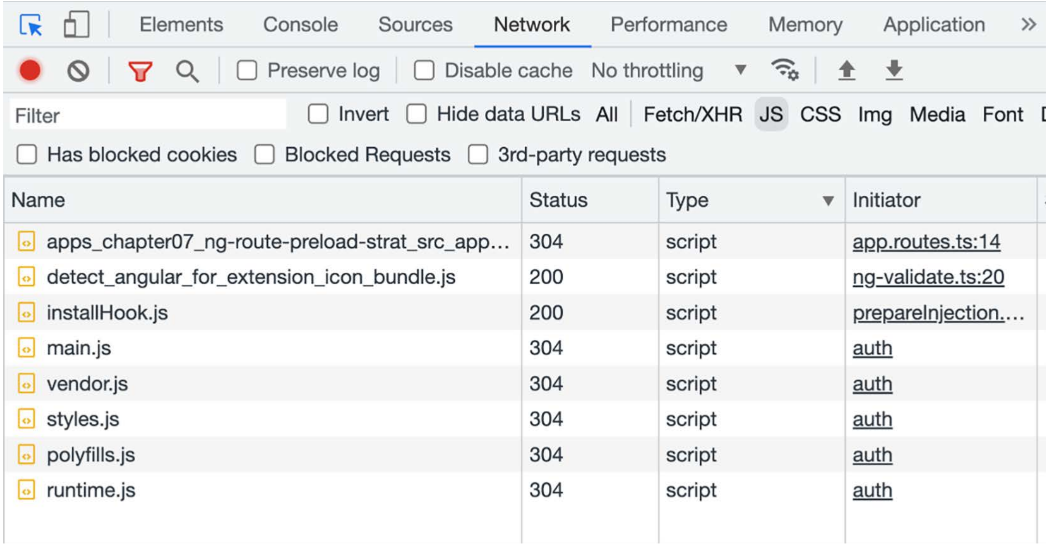
   

   No items in bucket. Add some fruits!

   Figure 7.7: The ng-route-preload-strat app running on http://localhost:4200

3. Open Chrome DevTools by pressing *Ctrl* + *Shift* + *C* on Windows or *Cmd* + *Shift* + *C* on Mac.

4. Navigate to the **Network** tab and filter on JavaScript files only. You should see something like this:


Elements
Console
Sources
Network
Performance
Memory
Application
☐ Preserve log
☐ Disable cache
No throttling
Filter
☐ Invert
☐ Hide data URLs
All
Fetch/XHR
JS
CSS
Img
Media
Font
☐ Has blocked cookies
☐ Blocked Requests
☐ 3rd-party requests

Name	Status	Type	Initiator
apps_chapter07_ng-route-preload-strat_src_app…	304	script	app.routes.ts:14
detect_angular_for_extension_icon_bundle.js	200	script	ng-validate.ts:20
installHook.js	200	script	prepareInjection.…
main.js	304	script	auth
vendor.js	304	script	auth
styles.js	304	script	auth
polyfills.js	304	script	auth
runtime.js	304	script	auth

Figure 7.8: JavaScript files loaded on app load

Now that we have the app running locally, let's see the next section for this recipe.

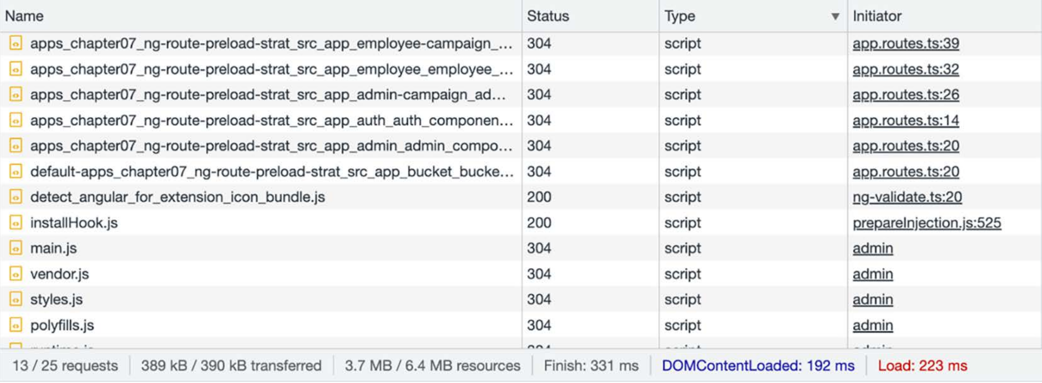
How to do it...

Notice in *Figure 7.8* the first network call. It is the JavaScript bundle that contains the `admin` component and the `bucket` component. Although the routes in `app-routes.ts` are all configured to be lazily loaded, we can still look into what happens if we use the `PreloadAllModules` strategy and then a custom preload strategy:

1. We're going to try out the `PreloadAllModules` strategy first. To use it, let's modify the `app/app.config.ts` file as follows:

```
import { ApplicationConfig } from '@angular/core';
import {
  PreloadAllModules,
  provideRouter,
  withEnabledBlockingInitialNavigation,
  withPreloading,
} from '@angular/router';
import { appRoutes } from './app.routes';
import { provideAnimations } from '@angular/platform-browser/animations';
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(
      appRoutes,
        withEnabledBlockingInitialNavigation(),
        withPreloading(PreloadAllModules)
      ),
      provideAnimations(),
    ],
  };
```

If you refresh the app, you should see all the JavaScript bundles being downloaded for all components, as follows:



Name	Status	Type	Initiator
apps_chapter07_ng-route-preload-strat_src_app_employee-campaign_...	304	script	app.routes.ts:39
apps_chapter07_ng-route-preload-strat_src_app_employee_employee_...	304	script	app.routes.ts:32
apps_chapter07_ng-route-preload-strat_src_app_admin-campaign_ad...	304	script	app.routes.ts:26
apps_chapter07_ng-route-preload-strat_src_app_auth_auth_componen...	304	script	app.routes.ts:14
apps_chapter07_ng-route-preload-strat_src_app_admin_admin_compo...	304	script	app.routes.ts:20
default-apps_chapter07_ng-route-preload-strat_src_app_bucket_bucke...	304	script	app.routes.ts:20
detect_angular_for_extension_icon_bundle.js	200	script	ng-validate.ts:20
installHook.js	200	script	prepareInjection.js:525
main.js	304	script	admin
vendor.js	304	script	admin
styles.js	304	script	admin
polyfills.js	304	script	admin

13 / 25 requests | 389 kB / 390 kB transferred | 3.7 MB / 6.4 MB resources | Finish: 331 ms | DOMContentLoaded: 192 ms | Load: 223 ms

Figure 7.9: All components being loaded eagerly using PreloadAllModules strategy

This already provides huge performance benefits because the minimal app `shell` is loaded first, and then all the other bundles are preloaded in parallel. This means that whenever the user navigates to a page now, they will not have to download the route's bundle and navigate right away (i.e., navigation will be faster). But what if we wanted to preload only the `admin` module, supposing our app is intended for admins mostly? We'll create a custom preload strategy for that.

2. Let's create a service named `CustomPreloadStrategy` by running the following command from within the `start` folder:

```
nx g service app-preload-strategy --project ng-route-preload-strat
```

3. In order to use our preload strategy service with Angular, our service needs to implement the `PreloadingStrategy` interface from the `@angular/router` package. Modify the newly created service as follows:

```
import { Injectable } from '@angular/core';
import { PreloadingStrategy } from '@angular/router';
@Injectable({
  providedIn: 'root'
})
export class CustomPreloadStrategyService implements PreloadingStrategy {
}
```

4. Next, we need to implement the `preload` method from the `PreloadingStrategy` interface for our service to work properly. Let's modify `CustomPreloadStrategyService` to implement the `preload` method, as follows:

```
import { Injectable } from '@angular/core';
import { PreloadingStrategy, Route } from '@angular/router';
import { Observable, of } from 'rxjs';
@Injectable({...})
export class CustomPreloadStrategyService implements PreloadingStrategy {
  preload(route: Route, load: () => Observable<any>):
    Observable<any> {
    return of(null)
  }
  logAndLoad(route: Route, load: () => Observable<any>) {
    console.log(`Preloading route: ${route.path}`);
    return load();
  }
}
```

5. Right now, our `preload` method returns `of(null)`. Instead, in order to decide which routes to preload, we're going to add an object to our route definitions as the `data` object that has a Boolean value for `admin` routes and `employee` routes to preload. Let's modify the `app-routes.ts` as follows:

```
...

export const appRoutes: Route[] = [
  { path: '', ...},
  { path: 'auth', ...},
  { path: 'admin', ..., data: { loadForAdmin: true } },
  {
    path: 'admin-campaign',
    ...,
    data: { loadForAdmin: true }
  },
  {
    path: 'employee',
    ...,
    data: { loadForEmployee: true },
  },
  {
    path: 'employee-campaign',
    ...,
    data: { loadForEmployee: true },
  },
];
```

6. Now let's add the logic to our `preload` method in the `AppPreloadStrategyService` class to work with the properties added in *step 5*. We're going to first inject the `AuthService` and will create some variables inside the `preload` method. Update the `app-preload-strategy.service.ts` file as follows:

```
...
import { AuthService } from './auth/auth.service';
import { UserType } from './constants/user-type';
...
export class AppPreloadStrategyService implements PreloadingStrategy {
  auth = inject(AuthService);
  preload(route: Route, load: () => Observable<any>): Observable<any> {
    const isLoggedIn = this.auth.isLoggedIn();
    if (!isLoggedIn) {
      return of(null)
    }
    const isAdmin = this.auth.loggedInUserType ===
      UserType.Admin;
    return of(null)
  }

  logAndLoad(route: Route, load: () => Observable<any>) {...}
}
```

7. Let's modify the `preload` method further to work with the `loadForAdmin` and `loadForEmployee` route data properties as follows:

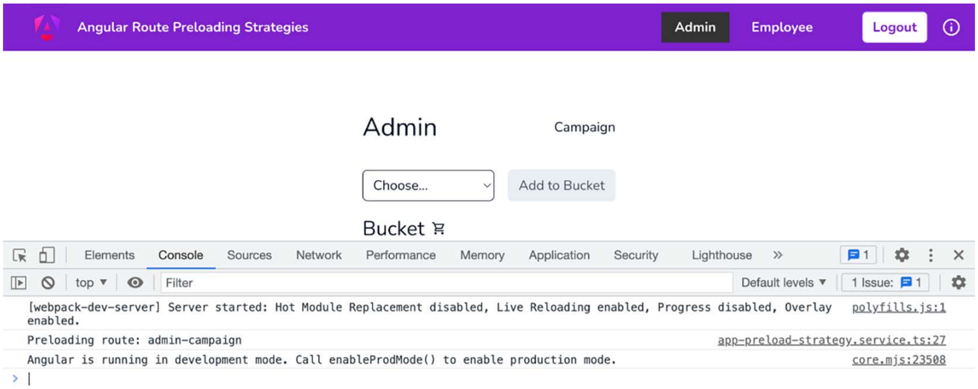
```
...
export class CustomPreloadStrategyService implements PreloadingStrategy {
...
  preload(route: Route, load: () => Observable<any>):
    Observable<any> {
    ...
    const isAdmin = this.auth.loggedInUserType ===
      UserType.Admin;
    if (
      (isAdmin && route.data?.['loadForAdmin']) ||
      (!isAdmin && route.data?.['loadForEmployee'])
    ) {
      return this.logAndLoad(route, load);
    }
```

```
      return of(null)
    }
  }
```

8. The final step is to use our custom preloading strategy. In order to do so, modify the `app/app.config.ts` file as follows:

```
import { ApplicationConfig } from '@angular/core';
import {
  provideRouter,
  withEnabledBlockingInitialNavigation,
  withPreloading,
} from '@angular/router';
import { appRoutes } from './app.routes';
import { AppPreloadStrategyService } from './app-preload-strategy.
service';
import { provideAnimations } from '@angular/platform-browser/animations';
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(
      appRoutes,
      withEnabledBlockingInitialNavigation(),
      withPreloading(AppPreloadStrategyService)
    ),
    provideAnimations(),
  ],
};
```

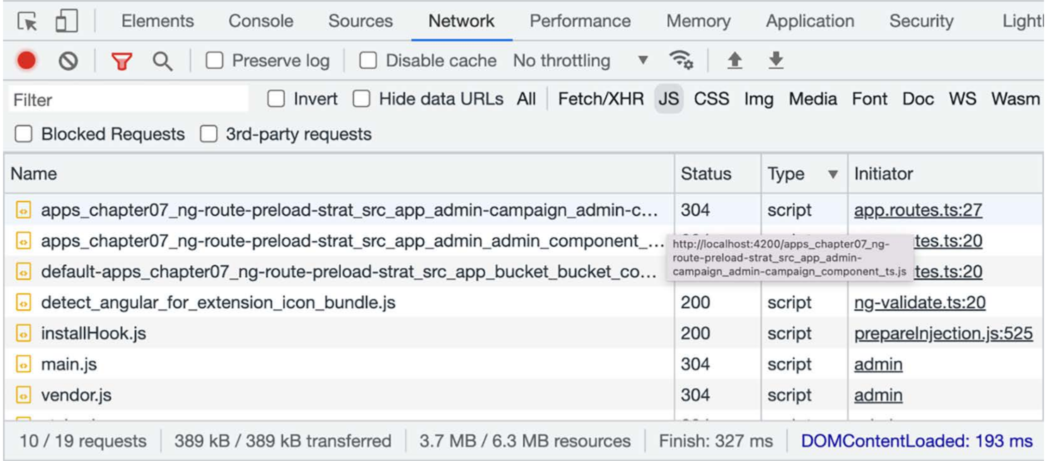
Voilà! If you refresh the app now and monitor the **Network** tab, you'll notice when you're logged in as `admin`, you can see in the console log that the `admin-campaign` route is preloaded, as shown in *Figure 7.10*:


Angular Route Preloading Strategies
Admin
Employee
Logout
Admin
Campaign
Choose...
Add to Bucket
Bucket
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
top
Filter
Default levels
1 Issue: 1

```
[webpack-dev-server] Server started: Hot Module Replacement disabled, Live Reloading enabled, Progress disabled, Overlay    polyfills.js:1
enabled.
Preloading route: admin-campaign                                                     app-preload-strategy.service.ts:27
Angular is running in development mode. Call enableProdMode() to enable production mode.                  core.mjs:23508
```

Figure 7.10: Preloading the admin-campaign route when landing on the admin route using the custom preload strategy

You can also have a look at the **Network** tab to see the bundles being downloaded:



Name	Status	Type	Initiator
apps_chapter07_ng-route-preload-strat_src_app_admin-campaign_admin-c…	304	script	app.routes.ts:27
apps_chapter07_ng-route-preload-strat_src_app_admin_admin_component_…			tes.ts:20
default-apps_chapter07_ng-route-preload-strat_src_app_bucket_bucket_co…			tes.ts:20
detect_angular_for_extension_icon_bundle.js	200	script	ng-validate.ts:20
installHook.js	200	script	prepareInjection.js:525
main.js	304	script	admin
vendor.js	304	script	admin

http://localhost:4200/apps_chapter07_ng-route-preload-strat_src_app_admin-campaign_admin-campaign_component_ts.js

10 / 19 requests | 389 kB / 389 kB transferred | 3.7 MB / 6.3 MB resources | Finish: 327 ms | DOMContentLoaded: 193 ms

Figure 7.11: JavaScript bundle for admin-campaign route preloaded

Try logging out and logging in as an employee. You'll see the `employee-campaign` route being preloaded. Not only that, if you refresh the employee-campaign route, you'll see the `employee` route being preloaded as well. The same goes for the admin-campaign and `admin` routes.

Now that you've finished the recipe, see the next section on how this works.

How it works…

Angular provides a great way to implement our own custom preloading strategy for our feature modules. We can decide easily which modules should be preloaded and which should not. In the recipe, we learned a very simple way to configure the preloading using the `data` object of the routes configuration by adding properties named `loadForAdmin` and `loadForEmployee`. We created our own custom preloading strategy service named `AppPreloadStrategyService`, which implements the `PreloadingStrategy` interface from the `@angular/router` package. The `PreloadingStrategy` class provides a method named `preload`, which has the following signature:

```
preload(route: Route, load: () => Observable<any>) => Observable<any>
```

The argument named `load` is a method from the Angular router that allows us to load the component/module by calling it. As a result, it provides an observable. Since the `preload` method must return an observable, we return `of(null)` when we do not want to preload a route, or we return `load` (which returns an observable) when we want to load the component associated with the route. The target in general is to have a custom preloading strategy that uses the `preload` method to decide whether a route should be preloaded. That is because Angular goes through each route using our custom preloading strategy and lets us decide which routes to preload. And that is it. We can see if the logged-in user is `admin` and has the `loadForAdmin` property in the route's data, or if the logged-in user is an employee and has the `loadForEmployee` property in the route's data. We then preload the route. Otherwise, we don't. And there's no preloading when the user is not logged in.

See also

- Route preloading strategies article on `web.dev`: `https://web.dev/route-preloading-in-angular/`
- Route preloading strategies in Angular: `https://www.youtube.com/watch?v=RQGLcMnh9k8`

Authorized access to routes using route guards

Not all routes in your Angular app should be accessible to everyone in the world. In this recipe, we'll learn how to create route guards in Angular to prevent unauthorized access to routes.

Getting ready

The app that we are going to work with resides in `start/apps/chapter07/ng-route-guards` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve ng-route-guards
   ```

   This should open the app in a new browser tab. You should see the following:

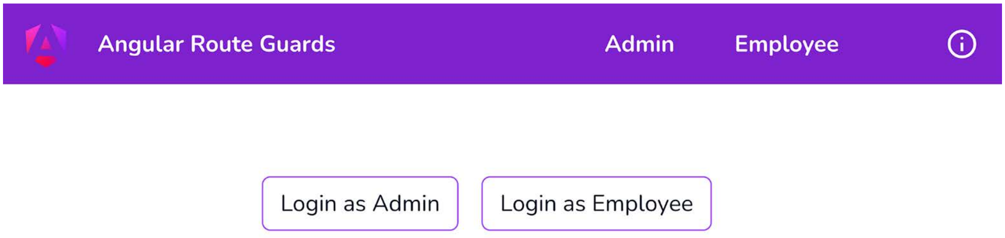


Figure 7.12: ng-route-guards app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

We have an app with a couple of routes already set up. You can log in as either an employee or an admin to get to the bucket list of the app. However, if you tap any of the two buttons in the header, you'll see that you can navigate to the **Admin** and **Employee** sections even without being logged in. This is what we want to prevent from happening. Notice in the `auth.service.ts` file that we already have a way for the user to log in, and we can check whether the user is logged in or not using the `isLoggedIn` method:

1. First, let's create a route guard that will only allow the user to go to the particular routes if the user is logged in. We'll name it `AuthGuard`. Let's create a new file inside the `auth` folder and name it `auth.guards.ts`. Then add the following code:

```
import { inject } from "@angular/core";
import { CanActivateFn, Router } from "@angular/router";
import { AuthService } from "./auth.service";

export const canActivateAdminOrEmployee: CanActivateFn =
  () => {
  const router = inject(Router);
  const authService = inject(AuthService);
  const isLoggedIn = authService.isLoggedIn();
  if (!isLoggedIn) {
    router.navigate(['/auth']);
    return false;
  }
  return true;
};
```

2. Now we can add this guard to the routes. Update `app.routes.ts` as follows:

```
      import { Route } from '@angular/router';
import { canActivateAdminOrEmployee } from './auth/auth.guards';
export const appRoutes: Route[] = [
  {...},
  { path: 'auth', ... },
  {
    path: 'admin',
    ...,
    canActivate: [canActivateAdminOrEmployee]
  },
  {
    path: 'employee',
    ...,
    canActivate: [canActivateAdminOrEmployee]
  }
];
```

If you try to go to the Admin or Employee pages without being logged in now, you'll see you are not able to do that anymore thanks to the route guard.

3. Let's make sure we can't go to the `auth` page if we're already logged in. Add a function guard in the `auth.guards.ts` file as follows:

```
...
import { UserType } from '../constants/user-type';
export const canActivateLogin: CanActivateFn = () => {
  const router = inject(Router);
  const authService = inject(AuthService);
  const isLoggedIn = authService.isLoggedIn();
  if (router.url === '/' && isLoggedIn) {
    const isAdmin = authService.loggedInUserType ===
      UserType.Admin;
    router.navigate([`/${isAdmin ? 'admin' :
      'employee'}`])
    return false
  }
  return !authService.isLoggedIn()
}

export const canActivateAdminOrEmployee: CanActivateFn = () => {...};
```

If you log in, and then click the browser's **Back** button, you'll see you can't go back to the `/auth` page either. Even if you open a new tab and go to `http://localhost:4200`, you'll see that it takes you to the correct page based on the logged-in user's type.

Great! You now are an authorization expert when it comes to guarding routes. With great power comes great responsibility. Use it wisely.

How it works...

Angular has switched to functional route guards since Angular v14.2. This makes it much easier to configure the routes compared to the previous version. There are many route guards, such as `CanActivate`, `CanDeactivate`, and `CanActivateChildren`. The `provideRouter` function used in `main.ts` makes it possible for us to provide the routes with functional route guards. A route can take the guards as an array against the guard property name. You can see how we provided our `canActivateAdminOrEmployee` guard against both the `/employee` and `/admin` routes in *step 2*. A functional guard is supposed to return a Boolean value or a `UrlTree`, a **promise** of Boolean or `UrlTree`, or an observable of Boolean or `UrlTree`. We've focused on the Boolean value's usage in our recipe. In the `canActivateAdminOrEmployee` guard, we check if the user is logged in. If that's true, we allow the route to be activated (i.e., we allow the routing to happen), or we navigate to the `/auth` route. In the `canActivateLogin` guard, we do something a bit more complex. Since someone could land on the home page (the route is `/`, which redirects to `/auth`), we have to first check if the user is logged in, and if that's the case, then which type of user this is. Based on evaluation, we route the user to either the `/employee` or `/admin` route.

See also

- Preventing unauthorized access in Angular routes: `https://angular.io/guide/router#preventing-unauthorized-access`
- Angular `CanActivateFn` docs: `https://angular.io/api/router/CanActivateFn`

Working with route parameters

Whether it is about building a REST API using Node.js or configuring routes in Angular, setting up routes is an absolute art, especially when it comes to working with parameters. In this recipe, you'll create some routes with parameters and will learn how to get those parameters in your components once the route is active.

Getting ready

The app that we are going to work with resides in `start/apps/chapter07/ng-route-params` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-route-guards
```

This should open the app in a new browser tab. You should see the following:

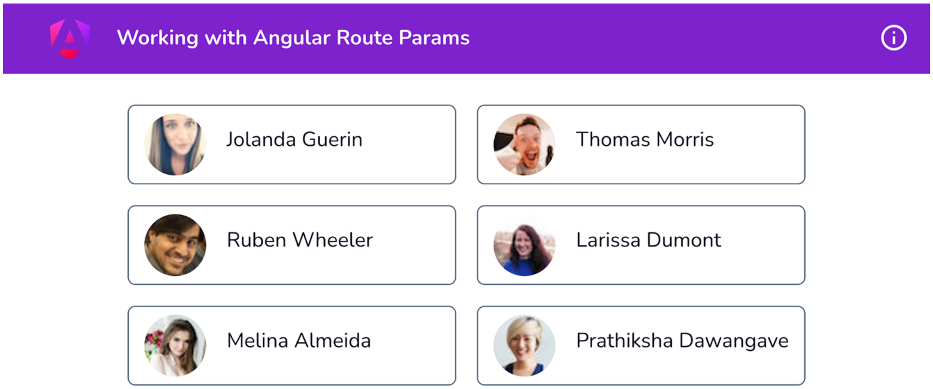


Figure 7.13: The ng-route-params app running on localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it…

The problem right now is that we have a route for opening the user details, but we don't see anything, i.e., we have a blank page. That's because this route doesn't have any idea about which user to show. Wouldn't it be nice to have a way to pass the clicked user's information from the users list page to the user detail page? That's exactly what we're going to do in this recipe:

1. First, we have to make our user route capable of accepting the `route` parameter. This is going to be a `required` parameter, which means the route will not work without passing this. Let's modify `app.routes.ts` to add this required parameter to the route definition, as follows:

```
import { Route } from '@angular/router';
export const appRoutes: Route[] = [
  { path: '', ...},
  { path: 'users', ...},
  {
    path: 'users/:uuid',
    loadComponent: () => ...,
  },
];
```

2. We'll now update the `src/app/components/users-list/users-list.component.html` file to change the links to use the `uuid` as follows:

```
<ul>
  <li
    routerLink="/users/{{user.uuid}}"
    *ngFor="let user of users">
    ...
  </li>
</ul>
```

We're now able to navigate to a particular user's route using the `uuid`, and you should also be able to see it in the address bar as follows:

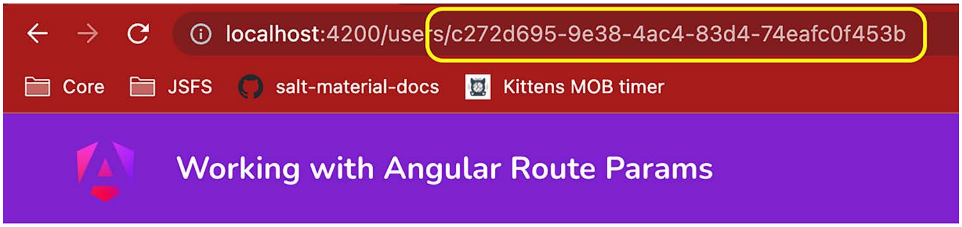

localhost:4200/users/c272d695-9e38-4ac4-83d4-74eafc0f453b
Core
JSFS
salt-material-docs
Kittens MOB timer
Working with Angular Route Params

Figure 7.14: The UUID being shown in the address bar

3. To get the user from `UserService` based on the `uuid`, we need to get the `uuid` value in `UserDetailsComponent` from the route parameters. Let's update the `user-details.component.ts` as follows:

```
...
import { ActivatedRoute, RouterModule } from '@angular/router';
import { filter, map, Observable} from 'rxjs';
...
@Component({...})
export class UserDetailsComponent {
  route = inject(ActivatedRoute);
  ...

  constructor() {
    this.user$ = this.route.paramMap.pipe(
      map((params) => params.get('uuid')),
      filter((uuid) => !!uuid),
      map((uuid) => {
        return this.userService.getById(uuid as string) ||
          null;
      })
    );
  }
}
```

You should be able to see the user we clicked from the `/users` page now. The only thing remaining is showing similar users.

4. Instead of creating another subscription, we can tap into the one we created in the previous step using the `tap` operator from RxJS. Update the `user-details.component.ts` file further, as follows:

```
...
import { filter, map, Observable, of, tap} from 'rxjs';
...

@Component({...})
export class UserDetailsComponent {
  ...
  constructor() {
    this.user$ = this.route.paramMap.pipe(
      map((params) => params.get('uuid')),
      filter((uuid) => !!uuid),
      tap((uuid) => {
        this.similarUsers$ =
          of(this.userService.getSimilar(uuid as string))
      }),
      map((uuid) => {...})
    );
  }
}
```

And boom, you should be able to see both the clicked user and similar users, as follows:

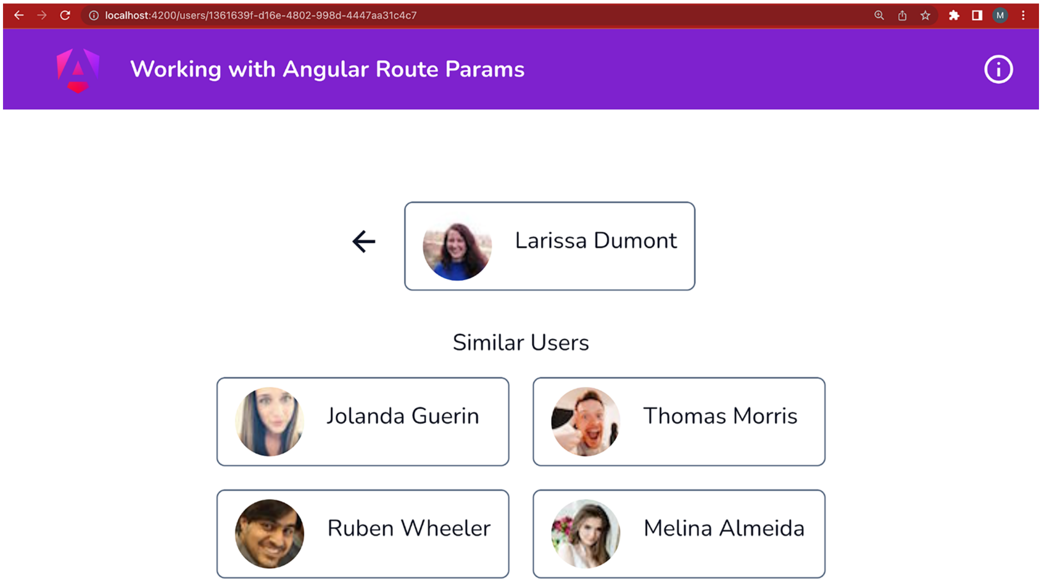

localhost:4200/users/1361639f-d16e-4802-998d-4447aa31c4c7
Working with Angular Route Params
Larissa Dumont
Similar Users
Jolanda Guerin
Thomas Morris
Ruben Wheeler
Melina Almeida

Figure 7.15: The UUID being shown in the address bar

Great! With this change, you can try refreshing the app on the home page and then click any user. You should see the current user as well as similar users being loaded. To understand all the magic behind the recipe, see the next section.

How it works...

It all begins when we change our route's path to `user/:userId`. This makes `userId` a parameter for our route. The other piece of the puzzle is to retrieve this parameter in `UserDetailsComponent` and then use it to get the target user, as well as similar users. For that, we use the `ActivatedRoute` service. The `ActivatedRoute` service holds a lot of necessary information about the current route and, therefore, we were able to fetch the current route's `uuid` parameter by subscribing to the `paramMap` observable, so even if the parameter changes while staying on a user's page, we still execute the necessary operations. The `ActivatedRoute` also has a `queryParamMap` observable, but it works for query parameters instead of route parameters. And if you don't want to subscribe to the `paramMap` observable and just want the value to appear once inside `ngOnInit` or in the constructor, you can also use the `snapshot` property on the `ActivatedRoute` object as follows:

```
this.route.snapshot.paramMap.get('uuid');
```

Notice that with the `paramMap` observable, we also use the `tap` operator to do something extra. That is assigning the value of similar users to the desired observable. Since both of the observables are only used in the template with the `async` pipe, there is no need to unsubscribe them ourselves as Angular takes care of it. Also, if you re-assign a new observable to a property being used by the `async` pipe, it automatically subscribes to the new one and cancels the previous subscription.

Note that you can also provide resolved values from the route definition directly to the component as `@Input()` properties using the `bindToComponentInputs` or `withComponentInputBinding` properties introduced in v16 for the Angular Router for Angular modules and standalone components, respectively. See the next section for relevant links including an article explaining the use of `withComponentInputBinding` as well.

See also

- *Accessing query parameters and fragments*: `https://angular.io/guide/router#accessing-query-parameters-and-fragments`
- *Getting route information* – Angular docs: `https://angular.io/guide/router#getting-route-information`
- *Bind Route Info to Component Inputs (New Router feature)*: `https://itnext.io/bind-route-info-to-component-inputs-new-router-feature-1d747e559dc4`

Showing a global loader between route changes

Building user interfaces that are snappy and fast is key to winning users. The apps become much more enjoyable for the end users, and it could bring a lot of value to the owners/creators of the apps. One of the core experiences on the modern web is to show a loader when something is happening in the background. In this recipe, you'll learn how to create a global user interface loader in your Angular app that shows whenever there is a route transition in the app.

Getting ready

The app that we are going to work with resides in `start/apps/chapter07/ng-global-loader` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-global-loader
```

This should open the app in a new browser tab and you should see the following:

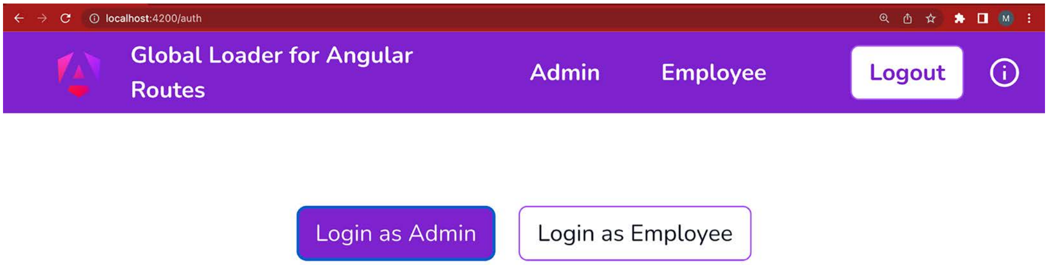


Figure 7.16: ng-global-loader app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

If you try logging in or logging out, you'll see that it takes a while before the route changes. We're simulating a delay in route change manually in the `canActivateLogin` route guard in the `start/apps/chapter07/ng-global-loader/src/app/auth/auth.guard.ts` file, which uses the `delay` operator from RxJS in an observable chain before checking if the user is logged in, and the type of user. The intention is to simulate the use cases where you need to route based on an HTTP call. In this recipe, we have the `LoaderComponent` already created, which we must use during the route changes:

1. We'll begin by adding a new property in the `app.component.ts` that says `isRouting`. We'll initialize it as `true` for now. We'll also import the `LoaderComponent` class in the `AppComponent` class's `imports` so we can use it in the template. Update the file as follows:

```
...
@Component({
  ...
  imports: [RouterModule, CommonModule, LoaderComponent],
  providers: [...],
})
export class AppComponent {
  ...
  router = inject(Router);
  isRouting = true;
  ...
}
```

2. Now we'll add the loader in the `app.component.html` to show it conditionally, and we'll wrap the entire app content in another variable. Update the file as follows:

```
<div *ngIf="isRouting; else app" class="loader-container
  fixed w-full h-full flex items-center justify-center ">
  <app-loader></app-loader>
</div>
<ng-template #app>
  <div class="toolbar" role="banner">...</div>
  <main class="content" role="main">...</main>
</ng-template>
```

   You should be able to see the loader consistently on the screen now, on every route. However, we want to make it work with the Angular Router.

3. We'll now update the `app.component.ts` file to listen to the router service's `events` property, and take an action upon the `NavigationStart` event. Modify the code in the `app.component.ts` file as follows:

```
import { NavigationStart, Router, RouterModule } from '@angular/router';
...
@Component({...})
export class AppComponent {
  constructor() {
    this.router.events.subscribe((event) => {
      if (event instanceof NavigationStart) {
        this.isRouting = true;
      }
    })
  }
  get isLoggedIn() {...}
}
```

If you refresh the app, you'll notice that `<app-loader>` still never goes away. That's because we're not marking the `isRouting` property as `false` anywhere.

4. To mark `isRouting` as `false`, we need to check for three different events: `NavigationEnd`, `NavigationError`, and `NavigationCancel`. Let's add some more logic to handle these three events and mark the property as `false`:

```
...
import { NavigationCancel, NavigationEnd, NavigationError,
NavigationStart, Router, RouterModule } from '@angular/router';
...

@Component({...})
export class AppComponent {
  ...
  isRouting = false;
  constructor() {
    this.router.events.subscribe((event) => {
      if (event instanceof NavigationStart) {
        this.isRouting = true;
      } else if (
        event instanceof NavigationEnd ||
        event instanceof NavigationError ||
        event instanceof NavigationCancel
      ) {
        this.isRouting = false;
      }
    })
  }
  ...
}
```

We now have a global loader that shows during the route navigation among different pages.

Congrats on finishing the recipe. You now can implement a global loader in Angular apps, which will show from the navigation start to the navigation end.

How it works...

The router service is a very powerful service in Angular. It has a lot of methods as well as observables that we can use for different tasks in our apps. For this recipe, we used the `events` observable. By subscribing to the `events` observable, we can listen to all the events that the `Router` service emits through the observable. One of the most common use cases for this (especially being a **Single-Page Application** (**SPA**)) is tracking page visits with Google Analytics or Mixpanel. For this recipe, we were only interested in the `NavigationStart`, `NavigationEnd`, `NavigationError`, and `NavigationCancel` events. The `NavigationStart` event is emitted when the router starts navigation. The `NavigationEnd` event is emitted when the navigation ends successfully. The `NavigationCancel` event is emitted when the navigation is canceled due to a route guard returning `false`, or redirects by using `UrlTree` due to some reason. The `NavigationError` event is emitted when there's an error due to any reason during the navigation. All of these events are of the `Event` type and we can identify the type of the event by checking whether it is an instance of the target event, using the `instanceof` keyword. Notice that since we had the subscription to the `Router.events` property in `AppComponent`, we didn't have to worry about unsubscribing the subscription because there's only one subscription in the app, and `AppComponent` will not be destroyed throughout the life cycle of the app.

See also

- Router events docs: `https://angular.io/guide/router#router-events`
- Router service docs: `https://angular.io/api/router/Router`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

8

Mastering Angular Forms

Getting user input is an integral part of almost any modern app that we use. Whether it is authenticating users, asking for feedback, or filling out business-critical forms, knowing how to implement and present forms to end users is always an interesting challenge. In this chapter, you'll learn about **Angular forms** and how you can create great user experiences using them.

Here are the recipes that we're going to cover in this chapter:

- Creating your first template-driven form with validation
- Creating your first reactive form with form validation
- Testing forms in Angular
- Server-side validation using asynchronous validator functions
- Implementing complex forms with reactive `FormArray`
- Writing your own custom form control using `ControlValueAccessor`

Technical requirements

For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter08`.

Creating your first template-driven form with validation

Let's start getting familiar with Angular forms in this recipe. In this one, you'll learn about the basic concepts of template-driven forms and will create a basic Angular form, using the template-driven forms API.

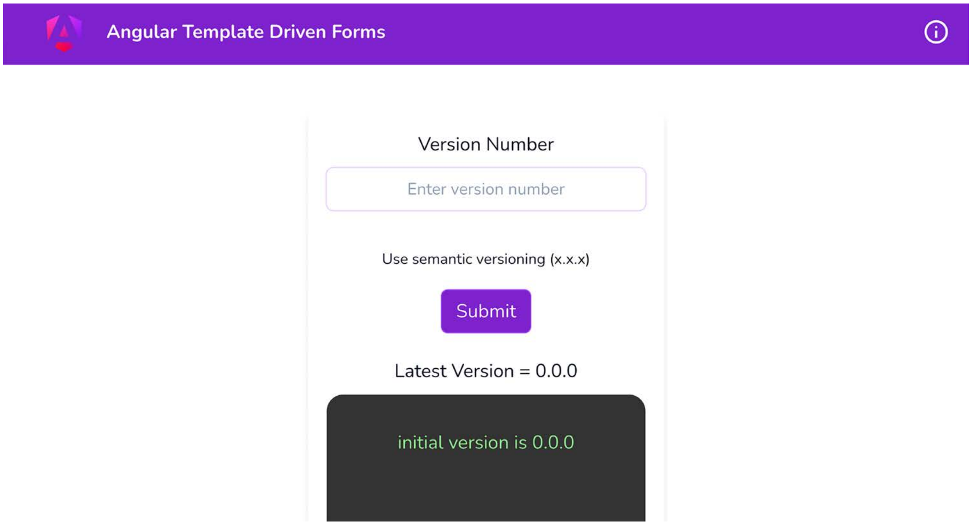
Getting ready

The app that we are going to work with resides in `start/apps/chapter08/ng-tdf` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-tdf
```

This should open the app in a new browser tab, and you should see the following:



Angular Template Driven Forms
Version Number
Enter version number
Use semantic versioning (x.x.x)
Submit

Latest Version = 0.0.0

initial version is 0.0.0

Figure 8.1: ng-tdf app running on http://localhost:4200

How to do it…

We have an Angular app that already has a bunch of components and some UI setup. We need to implement the functionality to add a new version to the logs. We'll use a template-driven form to allow the user to pick an app and submit a release version. Let's get started:

1. First, we'll add the `FormsModule` in the `src/app/components/version-control/version-control.component.ts` file in the project, as follows:

```
...
import { FormsModule } from '@angular/forms';
...
@Component({
  ...,
  imports: [CommonModule, FormsModule, VcLogsComponent]
})
export class VersionControlComponent  {...}
```

2. Now, we'll use a template-variable with `NgForm` and `NgModel` on the form input inside the component's HTML file to use template-driven forms. Update the `version-control.component.html` file as follows:

```
<form #versionForm="ngForm">
  <div class="form-group mb-4">
    <label for="versionNumber">Version Number</label>
    <input [(ngModel)]="versionInput" name="version"
      type="text" class="form-control my-2 text-center"
      id="versionNumber" aria-describedby="versionHelp"
      placeholder="Enter version number">
    ...
  </div>
  ...
</form>
```

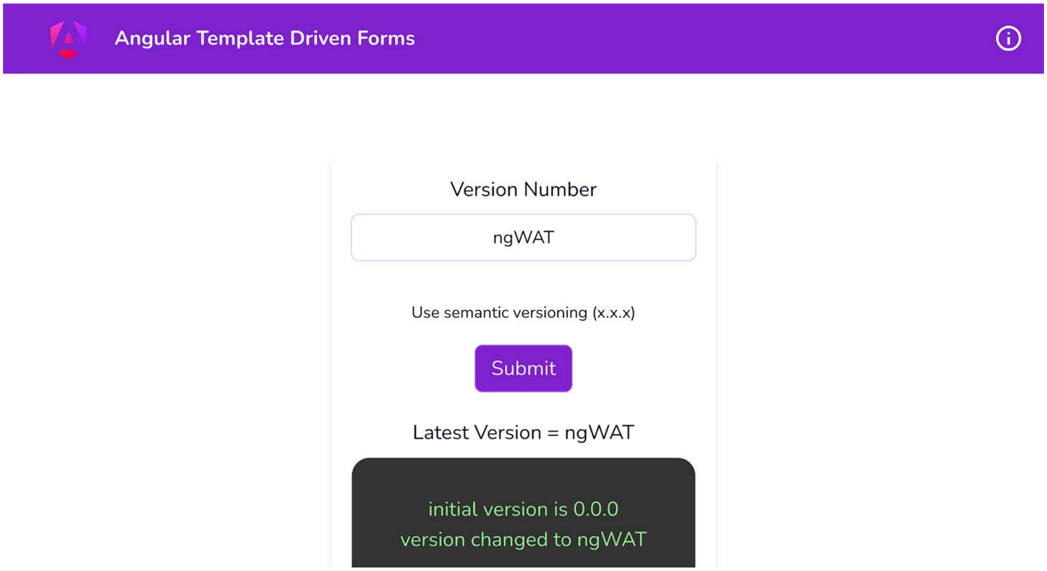
3. Let's create a method that will be triggered when the form is submitted. Update the `version-control.component.ts` file as follows:

```
...
import { FormsModule, NgForm } from '@angular/forms';
...
export class VersionControlComponent  {
  ...
  formSubmit(form: NgForm) {
    this.versionName = form.controls['version'].value;
  }
}
```

4. Now that we have the form submit handler, let's add the event listener to the template. Update the `version-control.component.html` file, as follows:

```
<form #versionForm="ngForm"
  (ngSubmit)="formSubmit(versionForm)">
  ...
</form>
<app-vc-logs [vName]="versionName"></app-vc-logs>
```

After adding this, you should be able to add new versions. However, you can use any value for the input or even provide nothing, and it will add as a new version. See the following screenshot for an example:


Angular Template Driven Forms
Version Number
ngWAT
Use semantic versioning (x.x.x)
Submit

Latest Version = ngWAT

initial version is 0.0.0
version changed to ngWAT

Figure 8.2: ng-tdf app without form validation

5. We'll add some form validation going forward. First of all, we'll make sure we only change the value of the `versionName` property when the form has a valid input. Update the `formSubmit` method in the `version-control.component.ts` file, as follows:

```
export class VersionControlComponent  {
  ...
  formSubmit(form: NgForm) {
    if (!form.valid) {
      return;
    }
    this.versionName = form.controls['version'].value;
  }
}
```

6. We will now add some validations in the template file. We'll make the required input and ensure that the version provided follows semantic versioning. Update the `version-control.component.html` file, as follows:

```
<form #versionForm="ngForm" (ngSubmit)="formSubmit(versionForm)">
  <div class="form-group mb-4">
    <label for="versionNumber">Version Number</label>
```

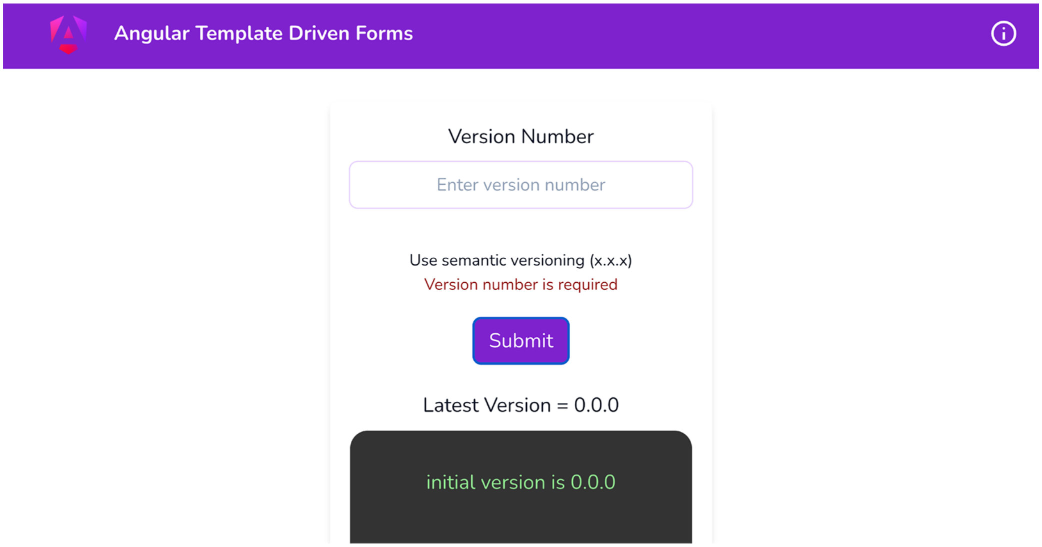
```
      <input [(ngModel)]="versionInput" pattern="(
        [0-9]+)\.([0-9]+)\.([0-9]+)" required name="version"
        type="text" class="form-control my-2 text-center"
        id="versionNumber" aria-describedby="versionHelp"
        placeholder="Enter version number">
      ...
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
  </form>
```

If you try submitting the form now with an invalid value, you'll see that the version provided doesn't get added to the logs, but if you provide a valid version like `2.1.0`, it will be added to the logs. However, this isn't great in terms of UX, since it doesn't tell us what's wrong.

7. Let's show some error messages based on the form validation we have. Update the `version-control.component.html`, as follows:

```
<form #versionForm="ngForm" (ngSubmit)="formSubmit(versionForm)">
  <div class="form-group mb-4">
    <label for="versionNumber">Version Number</label>
    <input [(ngModel)]="versionInput" pattern="(
      [0-9]+)\.([0-9]+)\.([0-9]+)" required name="version"
      type="text" class="form-control my-2 text-center"
      id="versionNumber" aria-describedby="versionHelp"
      placeholder="Enter version number">
    <small id="versionHelp" class="form-text text-muted
      block mt-6">Use semantic versioning (x.x.x)</small>
    <small class="error block" *ngIf="versionForm.submitted
      && versionForm.controls['version'].errors?
      .['required']">
      Version number is required
    </small>
    <small class="error block" *ngIf="versionForm.submitted
      && versionForm.controls['version']
      .errors?.['pattern']">
      Version number does not match the pattern (x.x.x)
    </small>
  </div>
  <button type="submit" class="btn btn-
    primary">Submit</button>
</form>
<app-vc-logs [vName]="versionName"></app-vc-logs>
```

If you now submit the form without any input, you should see the following error:


Angular Template Driven Forms
Version Number
Enter version number
Submit

Use semantic versioning (x.x.x)
Version number is required

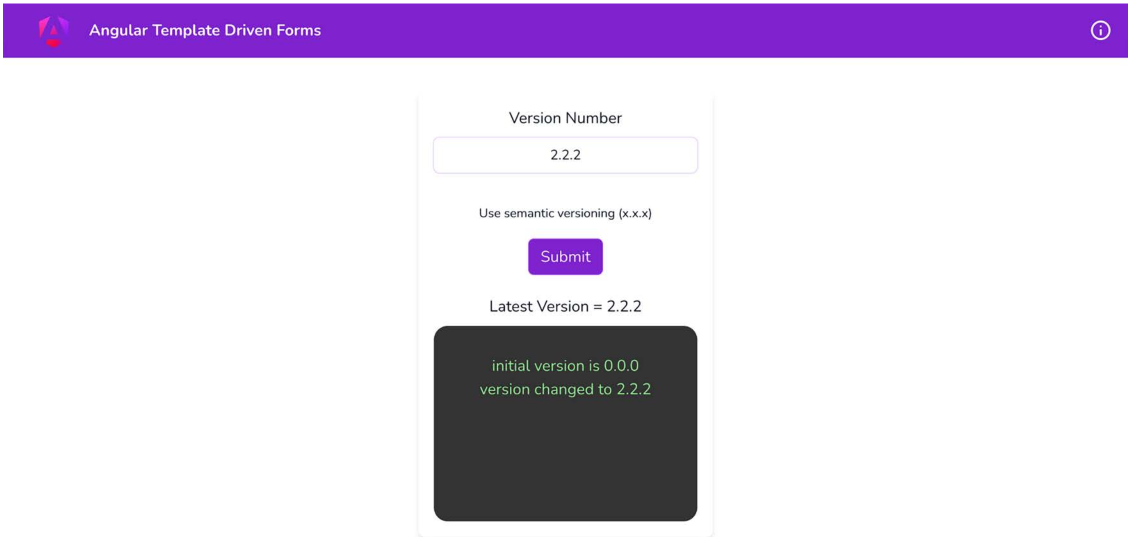
Latest Version = 0.0.0

initial version is 0.0.0

Figure 8.3: ng-tdf app with form validation

And if you try giving it the wrong value, you'll see a different error.

Great! Within a few minutes, we were able to create our first template-driven form in Angular with form validation. If you refresh the app now and add some versions, you should see the following:


Angular Template Driven Forms
Version Number
Submit

Version Number: 2.2.2

Use semantic versioning (x.x.x)

Latest Version = 2.2.2

initial version is 0.0.0
version changed to 2.2.2

Figure 8.4: ng-tdf app final output

Now that you know how the template-driven forms are created, let's see the next section to understand how it works.

How it works...

The key to using template-driven forms in Angular resides in `FormsModule`, the `ngForm` directive, and the `ngModel` binding. This would almost always include template variables along with the `name` attributes for inputs in the template. We began by adding the `FormsModule` in the `VersionControlComponent` class, which is necessary to use the `ngForm` directive and the `[(ngModel)]` two-way data binding. Then, we added the `[(ngModel)]` to the version input. We also added the `ngForm` attribute on the `form` element, similarly creating a `#versionForm` variable so that we can use it to see if the entire form is valid. Then, we added the `ngSubmit` handler to the `form` element so that we could trigger a method when the form is submitted. We also added the respective method named `formSubmit` in the `VersionControlComponent` class as well. Notice that we pass the `versionForm` variable to the `formSubmit` method, which makes it easier to test the functionality for us. Upon submitting the form, we use the form in the `formSubmit` method to get the value of the input and to create a new `version-log` item. Notice that we access the control/input by using `form.controls['version']`. The key version directly corresponds to the value of the `name` attribute on the input element in the template. `ngForm` automatically creates a `FormControl` instance for us, using the value of the `name` attribute of the input, and thus we can access it that way. Also, notice that if you provide an invalid version for the new release log, the app will show the respective errors based on either the empty input, or an invalid version format. This is because we have the `required` attribute set on the `<input />` element as well as the `pattern` attribute.

See also

- *Building a template-driven form in Angular*: `https://angular.io/guide/forms#building-a-template-driven-form`
- *Show and hide validation error messages* (Angular docs): `https://angular.io/guide/forms#show-and-hide-validation-error-messages`

Creating your first reactive form with validation

You've learned about template-driven forms in the previous recipe and are now confident in building Angular apps with them. Now, guess what? reactive forms are even better. Many known engineers and businesses in the Angular community recommend using only reactive forms for Angular apps. The reason is that when it comes to complex forms, Reactive forms make it easy to manage, i.e., you can decouple your form's logic from the template and add validators dynamically, etc. In this recipe, you'll build your first Reactive form and will learn its basic usage.

Getting ready

The app that we are going to work with resides in `start/apps/chapter08/ng-reactive-forms` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-reactive-forms
```

This should open the app in a new browser tab, and you should see the following:

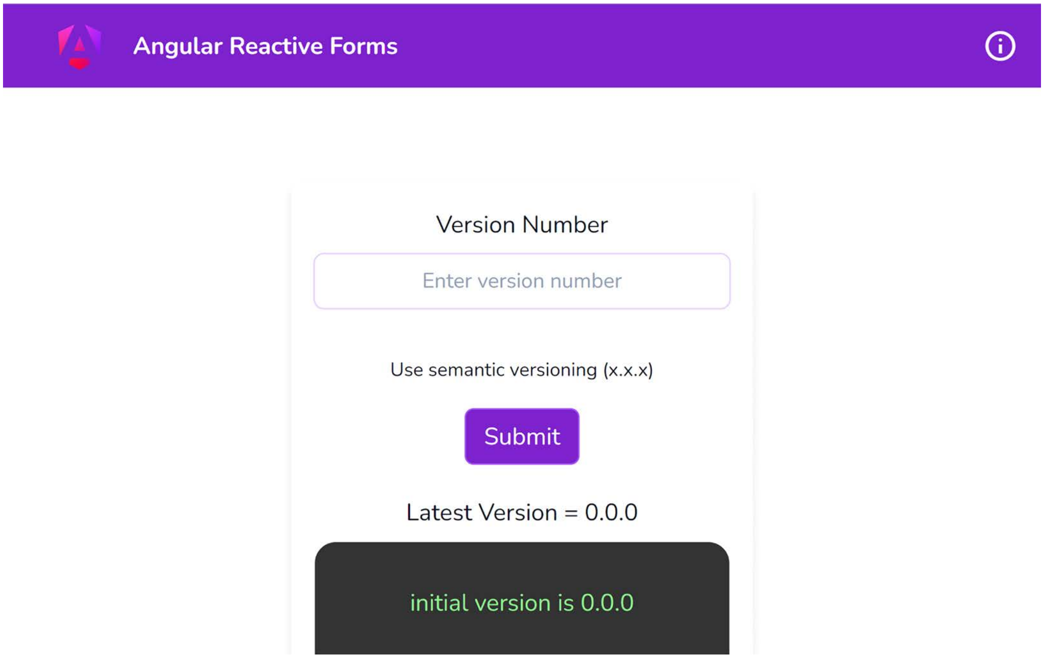

Angular Reactive Forms
Version Number
Enter version number
Use semantic versioning (x.x.x)
Submit
Latest Version = 0.0.0
initial version is 0.0.0

Figure 8.5: ng-reactive-forms app running on http://localhost:4200

Now that we have the app running locally, let's see the steps involved in this recipe in the next section.

How to do it...

So far, we have an app that has a `VcLogsComponent`, which shows a bunch of version logs that we create. We also have a `VersionControlComponent`, which has a form with which the release logs will be created. We now have to make our current form a Reactive form, using the reactive forms API. Let's get started:

1. First of all, we need to import `ReactiveFormsModule` into the imports of our `AppModule`. Let's do it by modifying the `app/components/version-control/version-control.component.ts` file, as follows:

```
...
import { ReactiveFormsModule } from '@angular/forms';
...
@Component({
  ...,
  imports: [CommonModule, VcLogsComponent,
    ReactiveFormsModule]
})
export class VersionControlComponent  {...}
```

2. We'll create the Reactive form shortly. First, import the required dependencies in the `version-control.component.ts` file, as follows:

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import {
  FormControl,
  FormGroup,
  ReactiveFormsModule,
  Validators,
} from '@angular/forms';
...
```

3. Now, we'll create a `FormGroup` in our `VersionControlComponent` class with a control named `version`. A `FormGroup` allows us to group multiple controls together in a form. Modify the `version-control.component.ts` file, as follows:

```
@Component({...})
export class VersionControlComponent {
  versionInput = '';
  versionName = '0.0.0';
  versionPattern = '([0-9]+)\.([0-9]+)\.([0-9]+)';
  versionForm = new FormGroup({
    version: new FormControl('', [
      Validators.required,
      Validators.pattern(this.versionPattern),
    ]),
  });
}
```

4. Now that we have the Reactive form in place, let's bind the `versionForm` in the template to the `<form>` element. Update the `version-control.component.html` file, as follows:

```
<form [formGroup]="versionForm">
  ...
</form>
```

5. Great! Now that we have the form group bound, we can also bind the `version` form control. Modify the `version-control.component.html` file further, as follows:

```
<form [formGroup]="versionForm">
  <div class="form-group mb-4">
    <label for="versionNumber">Version Number</label>
    <input formControlName="version" name="version"
      type="text" class="form-control my-2 text-center"
```

```
        id="versionNumber" aria-describedby="versionHelp"
        placeholder="Enter version number">
      ...
    </div>
    ...
  </form>
  ...
```

6. Let's decide what will happen when we submit this form. We'll call a method named `formSubmit` in the template and pass `versionForm` to it when the form is submitted. Modify the `version-control.component.html` file, as follows:

```
<form
  [formGroup]="versionForm"
  (ngSubmit)="formSubmit(versionForm)">
  ...
</form>
```

7. The `formSubmit` method doesn't yet exist. Let's create it now in the `VersionControlComponent` class. Modify the `version-control.component.ts` file, as follows:

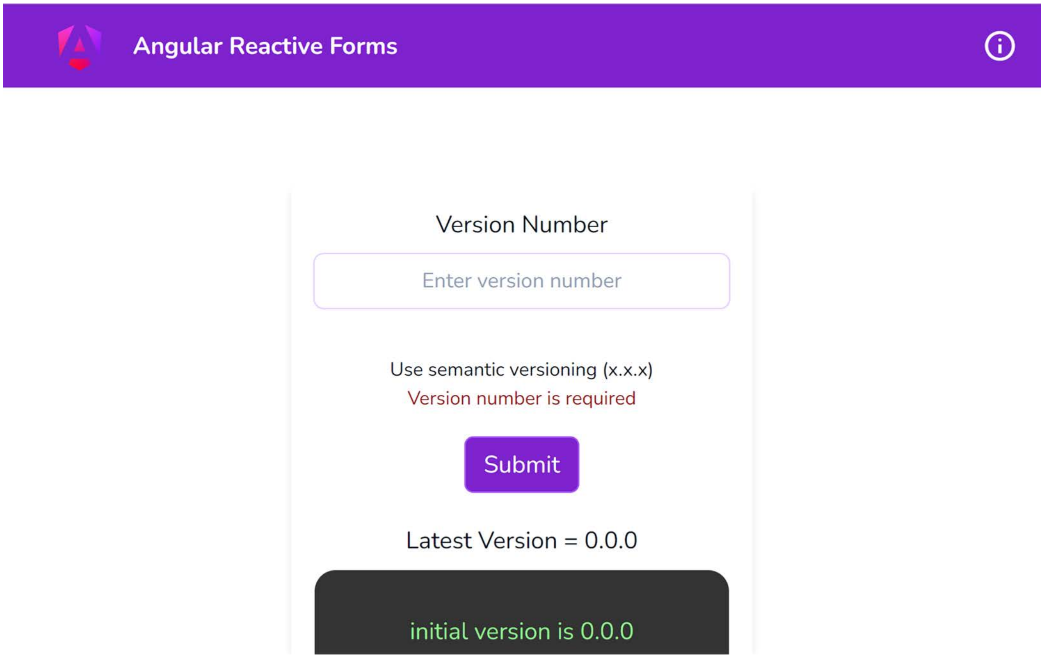
```
...
@Component({...})
export class VersionControlComponent {
  ...
  formSubmit(form: FormGroup) {
    if (!form.valid) {
      return;
    }
    this.versionName = form.controls['version'].value;
  }
}
```

If you try submitting the form now with an invalid value, you'll see that the version provided doesn't get added to the logs, but if you provide a valid version like `2.1.0`, it will be added to the logs. However, this isn't great in terms of UX, since it doesn't tell us what's wrong.

8. Let's show some error messages based on the form validation we have. Update the `version-control.component.html`, as follows:

```
<form [formGroup]="versionForm" (ngSubmit)="formSubmit(versionForm)">
  <div class="form-group mb-4">
    <label for="versionNumber">Version Number</label>
    ...
    <small class="error block" *ngIf="versionForm.dirty &&
      versionForm.controls.version.errors?.['required']">
      Version number is required
    </small>
    <small class="error block" *ngIf="versionForm.dirty &&
      versionForm.controls.version.errors?.['pattern']">
      Version number does not match the pattern (x.x.x)
    </small>
  </div>
  <button type="submit" class="btn btn-
    primary">Submit</button>
</form>
<app-vc-logs [vName]="versionName"></app-vc-logs>
```

If you now type something in the form and clear the input, you should see an error, as follows:


Angular Reactive Forms
Version Number
Enter version number
Submit

Use semantic versioning (x.x.x)

Version number is required

Latest Version = 0.0.0

initial version is 0.0.0

Figure 8.6: ng-reactive-forms app with form validation

And if you try giving it the wrong value, you'll see a different error.

Great! Within a few minutes, we were able to create our Reactive form in Angular with form validation. See the next section to understand how it works.

How it works...

To work with reactive forms in Angular, we need to import the `ReactiveFormsModule` in the components in the case of standalone components, and in the `NgModule` if you're using modules. In the recipe, we do exactly that, i.e., import the `ReactiveFormsModule` in the `VersionControlComponent` class's `imports` array. After that, we create a `FormGroup`, which is a Reactive form, and it takes an object as an argument for the constructor function. The object passed to the constructor can have further `FormControl`, nested form groups, and `FormArray`, etc. because they all inherit the `AbstractControl` class, as shown in *Figure 8.6*.

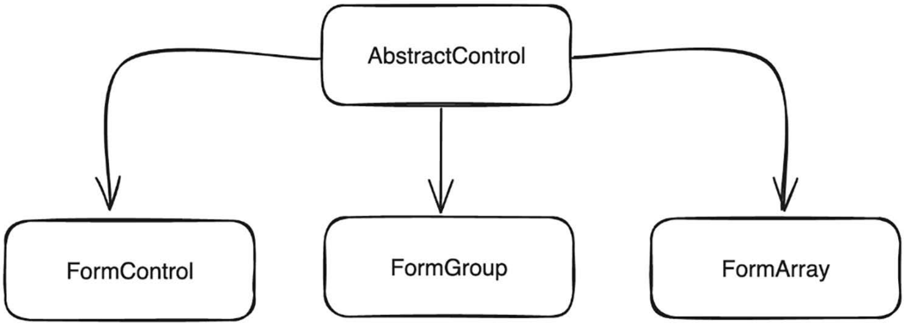


Figure 8.7: Base class AbstractControl being inherited

In our recipe, we provide a key named `version` against a `FormControl`. The `FormControl` constructor method takes two arguments in our recipe; the first one is the default value, and the second is an array of `Validators`. Notice that we're using the following validators:

```
Validators.required
Validators.pattern
```

Also, notice that the pattern we're using to validate semantic versioning is `([0-9]+)\.([0-9]+)\.([0-9]+)`. We then use the `FormGroup` (i.e., the `versionForm` variable) in the template on the `<form>` element, using the `[formGroup]` binding. We then bind the `version` `FormControl` in the template using the `formControlName` attribute. We handle the form submission using the `(ngSubmit)` listener on the `<form>` element. In the `VersionControlComponent` class, we have the `formSubmit` method to handle the form submission. This is also what we bind the event listener to in the template. When this method is called, we check if the form is valid.

The validation on each `FormGroup`, `FormArray`, or `FormControl` happens on the `change`, `blur`, or `submit` event. And since they are inherited from the `AbstractControl` class, they can be configured via the `updateOn` property on the `AbstractControl`. In our case, the validation happens on `change` by default. If the form is not valid, we simply do not do anything. Otherwise, we set the value of the `versionName` property, which adds a new log to the logs list.

Finally, we show the errors based on the form validation in the template, using the `errors` property on the `version` form control. When the form is empty after making any initial change to it, the `errors` object on the form control gets the `required` property set to `true`.

When the user adds a value to the input, the `required` property is removed, and the validation of `pattern` gets triggered, and if the pattern `([0-9]+)\.([0-9]+)\.([0-9]+)` doesn't match the value of the input, we get the `pattern` property added to the form control's `errors` object. Notice that in *step 8*, we use these properties from the `errors` object to show the relevant error.

See also

- Angular's guide to reactive forms: `https://angular.io/guide/reactive-forms`

Testing forms in Angular

To make sure we build robust and bug-free forms for end users, it is a really good idea to have tests relating to your forms. It makes code more resilient and less prone to errors. In this recipe, you'll learn how to test your template-driven forms using unit tests.

Getting ready

The app that we are going to work with resides in `start/apps/chapter08/ng-testing-forms` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project along with the backend server:

   ```
   npm run serve ng-testing-forms with-server
   ```

   This should open the app in a new browser tab, and you should see the following:

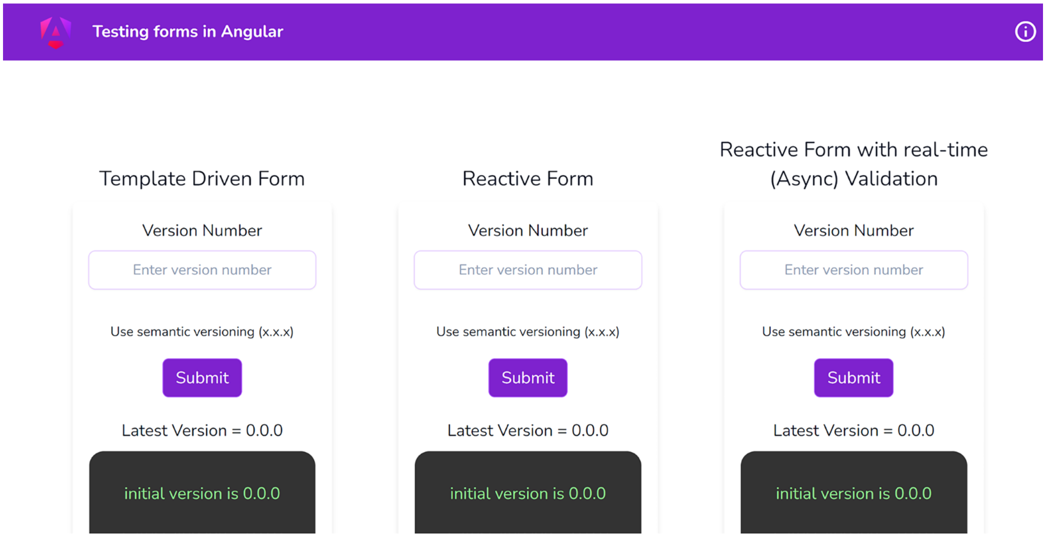

Testing forms in Angular
Template Driven Form
Reactive Form
Reactive Form with real-time (Async) Validation
Version Number
Enter version number
Use semantic versioning (x.x.x)
Submit
Latest Version = 0.0.0
initial version is 0.0.0

Figure 8.8: ng-testing-forms app running on http://localhost:4200

Now that we have the app running locally, let's see the steps involved in this recipe in the next section.

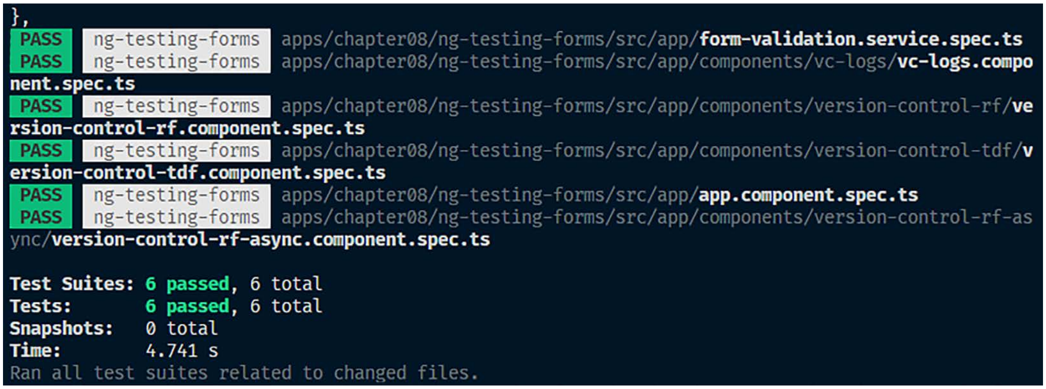
How to do it...

We have an app that contains three different components. The first one has a template-driven form, the second one has a reactive form with validations, and the third has a reactive form with both synchronous and asynchronous validations. The implementation of all three of them is identical, except the third component's validation kicks in as soon as the form is dirty instead of waiting to be submitted. The form also has validations applied to the inputs. Let us start looking into how we can test this form:

1. First, run the following command to run the unit tests:

   ```
   npm run test ng-testing-forms
   ```

   Once the command is run, you should see all the tests passing on the console as follows:

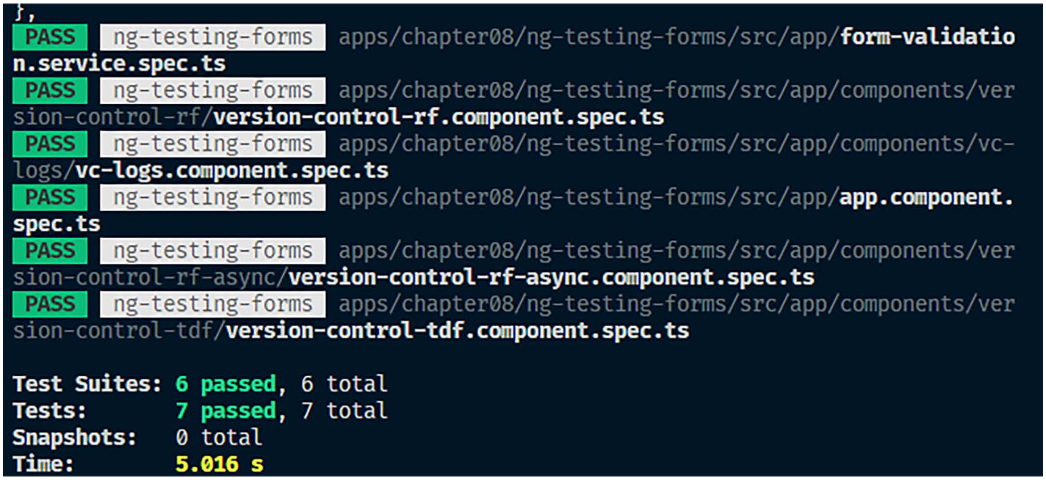
   ```
   },
    PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/form-validation.service.spec.ts
    PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/components/vc-logs/vc-logs.compo
   nent.spec.ts
    PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/components/version-control-rf/ve
   rsion-control-rf.component.spec.ts
    PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/components/version-control-tdf/v
   ersion-control-tdf.component.spec.ts
    PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/app.component.spec.ts
    PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/components/version-control-rf-as
   ync/version-control-rf-async.component.spec.ts

   Test Suites: 6 passed, 6 total
   Tests:       6 passed, 6 total
   Snapshots:   0 total
   Time:        4.741 s
   Ran all test suites related to changed files.
   ```

   Figure 8.9: Unit tests running with ng-testing-forms

2. Let us write the tests for the template-driven form. This is easy. We will update the `version-control-tdf.component.spec.ts` file, as follows:

   ```
   ...
   describe('VersionControlTdfComponent', () => {
     ...
     it('should create', () => {...});

     it('should submit the form with valid version', () => {
       component.versionForm.controls['version']
         .setValue('2.2.4');
       fixture.debugElement.nativeElement.querySelector('button').click();
       expect(component.versionName).toBe('2.2.4');
     });
   ```

If you look at the tests now, you should see all the tests passing as follows:



```
},
 PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/form-validatio
n.service.spec.ts
 PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/components/ver
sion-control-rf/version-control-rf.component.spec.ts
 PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/components/vc-
logs/vc-logs.component.spec.ts
 PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/app.component.
spec.ts
 PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/components/ver
sion-control-rf-async/version-control-rf-async.component.spec.ts
 PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/components/ver
sion-control-tdf/version-control-tdf.component.spec.ts

Test Suites: 6 passed, 6 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        5.016 s
```

Figure 8.10: First test for template-driven forms passing

3. Let's add another test to the file. This will test if we are able to see the error message of the required input. Let's update the `version-control-tdf.component.spec.ts` file, as follows:

```
it('should show required error', () => {
  component.versionForm.controls['version']
   .setValue('2.2.4');
  fixture.detectChanges();
  component.versionForm.controls['version'].setValue('');
  fixture.detectChanges();
  fixture.debugElement.nativeElement
   .querySelector('button').click();
  fixture.detectChanges();
  expect(component.versionName).toBe('0.0.0');
  expect(
   fixture.debugElement.nativeElement
    .querySelector('.error')
    .textContent.trim()
  ).toBe('Version number is required');
 });
```

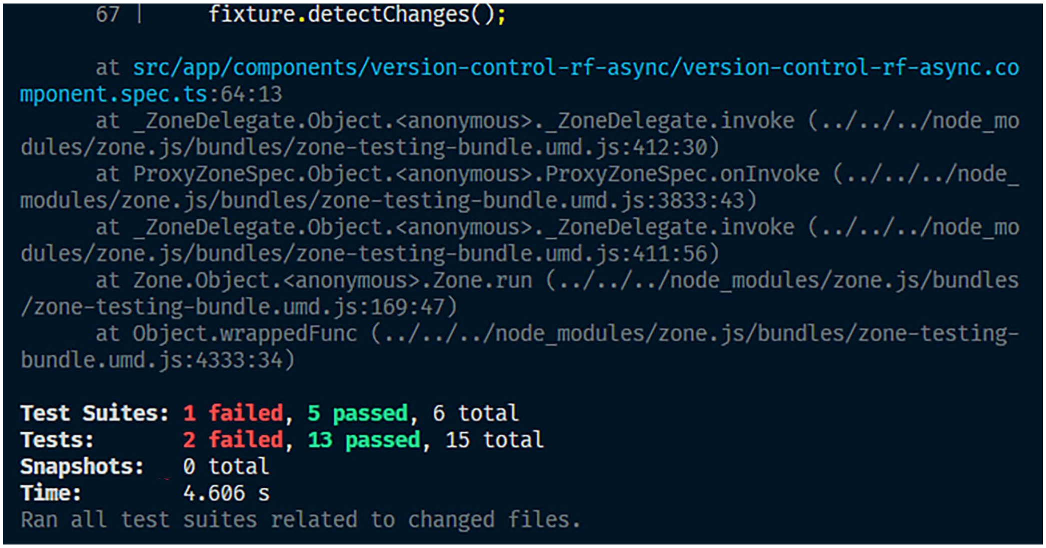
4. Let's add one more test. This will test if we are able to see the message for the pattern of the version being valid. Add the following test to the same file:

   ```
   it('should show pattern error', () => {
       component.versionForm.controls['version']
         .setValue('2.2.4');
       fixture.detectChanges();
       component.versionForm.controls['version']
         .setValue('invalid input');
       fixture.detectChanges();
       fixture.debugElement.nativeElement
         .querySelector('button').click();
       fixture.detectChanges();
       expect(component.versionName).toBe('0.0.0');
       expect(
         fixture.debugElement.nativeElement
           .querySelector('.error')
           .textContent.trim()
       ).toBe('Version number does not match the pattern (x.x.x)');
     });
   ```

   Great! You should see all the tests still passing. The fun fact is that the tests are identical for the Reactive form (with synchronous validators only) example as well.

5. Copy all the three tests we added from *step 2* to *step 4* to the file `version-control-rf.component.spec.ts`.
6. Now comes the tricky part, i.e., working with `async` validators. We cannot just copy and paste the tests for this one and expect the tests to pass when we have an incorrect version provided in the form. We need some additional work to do in this case. First, copy the same tests from the `version-control-tdf.component.spec.ts` file into the `version-control-rf-async.component.spec.ts` file.

   You will notice that we have two tests failing now, as follows:



```
      67 |      fixture.detectChanges();

      at src/app/components/version-control-rf-async/version-control-rf-async.co
mponent.spec.ts:64:13
      at _ZoneDelegate.Object.<anonymous>._ZoneDelegate.invoke (../../../node_mo
dules/zone.js/bundles/zone-testing-bundle.umd.js:412:30)
      at ProxyZoneSpec.Object.<anonymous>.ProxyZoneSpec.onInvoke (../../../node_
modules/zone.js/bundles/zone-testing-bundle.umd.js:3833:43)
      at _ZoneDelegate.Object.<anonymous>._ZoneDelegate.invoke (../../../node_mo
dules/zone.js/bundles/zone-testing-bundle.umd.js:411:56)
      at Zone.Object.<anonymous>.Zone.run (../../../node_modules/zone.js/bundles
/zone-testing-bundle.umd.js:169:47)
      at Object.wrappedFunc (../../../node_modules/zone.js/bundles/zone-testing-
bundle.umd.js:4333:34)

Test Suites: 1 failed, 5 passed, 6 total
Tests:       2 failed, 13 passed, 15 total
Snapshots:   0 total
Time:        4.606 s
Ran all test suites related to changed files.
```

Figure 8.11: Tests failing for a form with asynchronous validators, using a service

7. We are going to mock the `FormValidationService` for our tests. Update the `version-control-rf-async.component.spec.ts`, as follows:

```
import { provideHttpClient } from '@angular/common/http';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { of } from 'rxjs/internal/observable/of';
import { FormValidationService } from '../../form-validation.service';

import { VersionControlRfAsyncComponent } from './version-control-rf-
async.component';

const FormValidationServiceMock = {
  versionValidator() {
    return () => of(null);
  },
};
```

8. Let us provide our mock service to the `TestBed` for the tests in the same file, as follows:

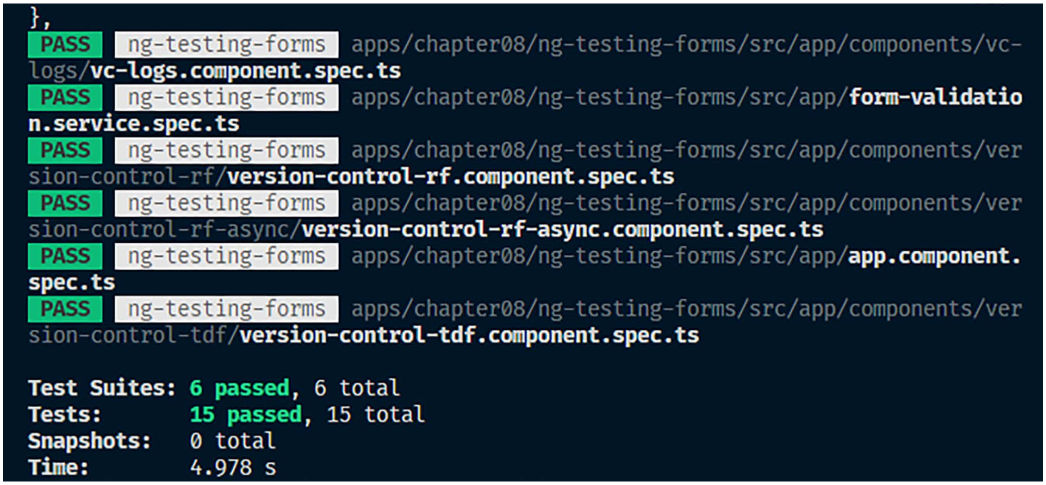
```
...
describe('VersionControlRfAsyncComponent', () => {
 ...
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [VersionControlRfAsyncComponent],
      providers: [
        provideHttpClient(),
        {
          provide: FormValidationService,
          useValue: FormValidationServiceMock,
        },
      ],
    }).compileComponents();
    ...
  });
});
```

With the above change, the tests `'should submit the form with valid version'` and `'should show required error'` should pass. Yay! However, the last test failed. We will fix that.

9. Let's update the final failing test. We are going to update our `FormValidationServiceMock` object to return a `pattern` error if there is an invalid input. Update the file as follows:

```
...
import { AbstractControl } from '@angular/forms';
import { of } from 'rxjs/internal/observable/of';
import { FormValidationService } from '../../form-validation.service';

import { VersionControlRfAsyncComponent } from './version-control-rf-
async.component';

const FormValidationServiceMock = {
  versionValidator(control: AbstractControl) {
    return () => {
      if (control.value === 'invalid input') {
        return of({ pattern: true });
      }
      return of(null);
    };
  }};
```

And voila! All of the tests should be passing now, as follows:



```
},
 PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/components/vc-
logs/vc-logs.component.spec.ts
 PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/form-validatio
n.service.spec.ts
 PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/components/ver
sion-control-rf/version-control-rf.component.spec.ts
 PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/components/ver
sion-control-rf-async/version-control-rf-async.component.spec.ts
 PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/app.component.
spec.ts
 PASS   ng-testing-forms  apps/chapter08/ng-testing-forms/src/app/components/ver
sion-control-tdf/version-control-tdf.component.spec.ts

Test Suites: 6 passed, 6 total
Tests:       15 passed, 15 total
Snapshots:   0 total
Time:        4.978 s
```

Figure 8.12: All tests passing for the recipe

Awesome! You now know a bunch of techniques to test your Angular forms. However, some of these techniques might still require some explanation. See the next section to understand how it all works.

How it works...

Testing Angular forms can be a bit of a challenge, as it depends on how complex the form is, what use cases you want to test, and how complex those use cases are. This is also true for cases where you have several dependencies to work with. In our recipe, we started with the template-driven forms, and it was easy for us to test them, since the component has a `ViewChild()` of the type `NgForm`. This makes it easier to write our first test that sets the value of the `version` form control. Then, we click the **Submit** button and expect the component's `versionName` property value to be the one we entered in the form. Easy peasy!

For the second test, we first set the value of the form to `2.2.4`, and then we empty the form by setting the value to an empty string. Then, we submit the form and expect the value of the component's `versionName` property to remain unchanged and still be `0.0.0`. However, what we also check is that, inside the form, we can see the error that says, `'Version number is required'`. Notice that we use the `fixture.debugElement.nativeElement.querySelector` method to get the desired error element, and then we check its `textContent` value.

For the test to check the pattern, we do the same as for the second test, but instead of emptying the input, we provide `'invalid input'` as the new value to the form control. The template-driven form picks it and shows the error `'Version number does not match the pattern (x.x.x)'`, which is what we expect in the test as well.

The same three tests work exactly the same as the `Reactive Forms` example we have. The difference this time is that the `versionForm` property in the component is not `NgForm` but, rather, a `FormGroup`. It is amazing that Angular has the same API (i.e., the same methods) in both `NgForm` and `FormGroup` to set and retrieve the values. This is because it creates a top-level `FormGroup` behind the scenes, thus making our tests the same.

For the tests with `async` validators, we did something special. Apart from copying and pasting the tests from the other examples, we had to mock the `FormValidationService`. This is because this example had an asynchronous validator function, which comes from the `FormValidationService` class, and it is a general best practice to mock all the dependencies of a unit so that we can purely test it.

We start by creating a stub named `FormValidationServiceMock` (we'll learn more about stubs in *Chapter 10 ,Writing Unit Tests in Angular with Jest*). Then, we provide this stub in the `TestBed` against the `FormValidationService` class. Finally, we make sure that our `versionValidator` method in the stub returns the correct error when the input is not valid. And that's it. Without many changes in the tests themselves, we are able to test all three kinds of forms with minor adjustments. I hope you learned quite a few techniques in this recipe.

See also

- *Testing template-driven forms*: `https://angular.io/guide/forms-overview#testing-template-driven-forms`

Server-side validation using asynchronous validator functions

Form validations are straightforward in Angular, with the reason being the super-awesome validators that Angular provides out of the box. These validators are synchronous, meaning that as soon as you change the input, the validators kick in and provide you with information about the validity of the values right away. However, sometimes, you might rely on some validations from a backend API or have some asynchronous logic that you need to execute to validate the form value. These situations would require something called asynchronous validators. In this recipe, you're going to create your first asynchronous validator.

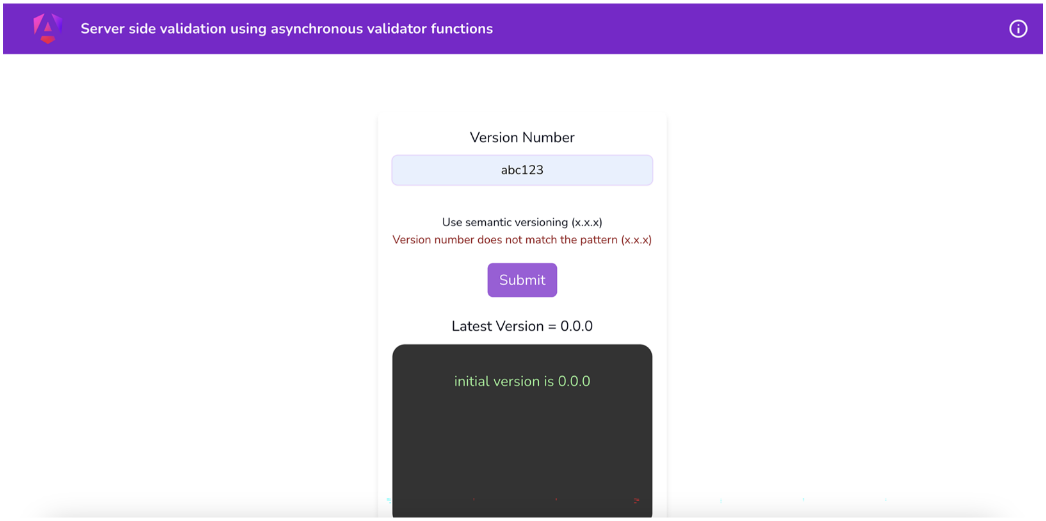
Getting ready

The app that we are going to work with resides in `start/apps/chapter08/ng-rf-async-validator` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project along with the backend server:

   ```
   npm run serve ng-rf-async-validator with-server
   ```

   This should open the app in a new browser tab, and you should see the following:



Server side validation using asynchronous validator functions
Version Number
abc123
Use semantic versioning (x.x.x)
Version number does not match the pattern (x.x.x)
Submit
Latest Version = 0.0.0
initial version is 0.0.0

Figure 8.13: Asynchronous validators app running on http://localhost:4200

Now that we have the app running, let's see the steps involved in this recipe in the next section.

How to do it...

We already have the backend with an endpoint that can accept a version as a query parameter, returning a `boolean` for if the provided version is a valid version according to the pattern (`x.x.x`) and is a newer version than the previously provided one. We'll create an `async` validator to make sure that the new releases have a valid version. Let's begin:

1. First, we will modify the `VersionService` class in the `version.service.ts` file, as follows, to add a method to validate the form input via the `http://localhost:3333/api/version/validate` endpoint:

```
...
export class VersionService {
  ...
  validateVersion(version: string): Observable<{ error:
    string }> {
    return this.http.get<{ error: string }>(
      `${this.apiBaseUrl}/validate?val=${version}`
    );
  }

  submitVersion(version: string): Observable<{ success:
    boolean }> {...);
  }
}
```

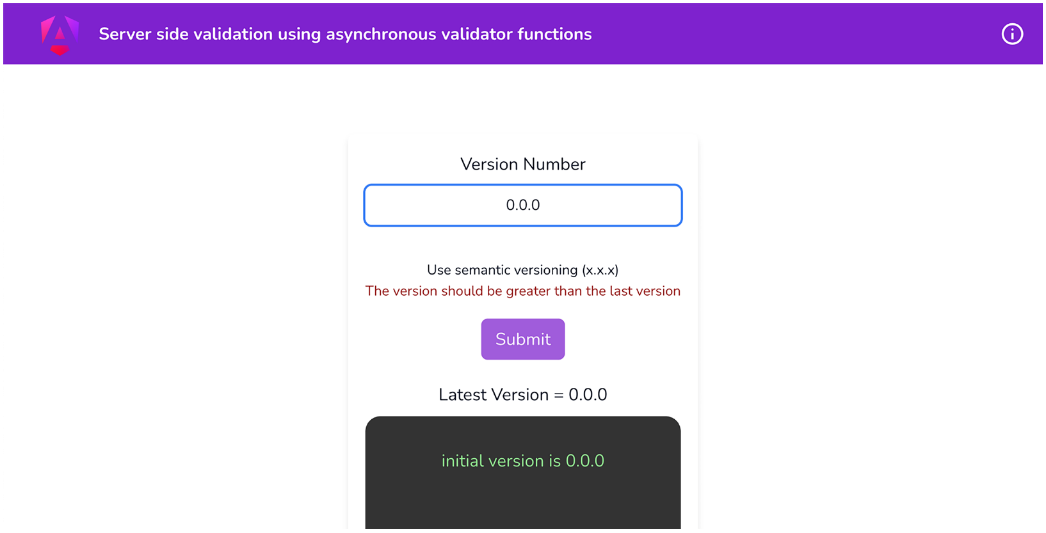
2. We will now create an **Async Validator Function** (`AsyncValidatorFn`) inside the `VersionService` class. This is something we will bind later to the form in the app. Let us update the `version.service.ts` file, as follows:

```
import { HttpClient } from '@angular/common/http';
import { Injectable, inject } from '@angular/core';
import { Observable } from 'rxjs/internal/Observable';
import { AbstractControl, AsyncValidatorFn, ValidationErrors } from '@
angular/forms';
import { timer, switchMap, map } from 'rxjs';
...
export class VersionService {
  ...



  versionValidator(): AsyncValidatorFn {
    return (control: AbstractControl): Observable<
      ValidationErrors | null > => {
      return timer(500).pipe(
        switchMap(() => this.validateVersion(
         control.value)),
        map(({ error }) => {
          const errors: ValidationErrors = {};
          if (error === null) {
            return null;
          }
          errors[error] = true;
          return errors;
        })
      );
    };
  }
  ...
}
```

3. We will use the `versionValidator` method we just created inside the `VersionControlComponent` class with the `versionForm`. To do that, let's modify the `version-control.component.ts` file, as follows:

```
import { Component, OnInit, inject } from '@angular/core';
...
export class VersionControlComponent implements OnInit {
  ...
  ngOnInit() {
    this.versionForm.controls.version.setAsyncValidators(
      this.versionService.versionValidator()
    );
  }

  formSubmit(form: FormGroup) {...}
}
```

If you now try submitting a version equal to or lower than the last submitted one, you will receive an error, as follows:



Server side validation using asynchronous validator functions

Version Number

0.0.0

Use semantic versioning (x.x.x)

The version should be greater than the last version

Latest Version = 0.0.0

initial version is 0.0.0

Figure 8.14: Error being shown when a lower or equal version number is provided

Cool! So, you now know how to create an asynchronous validator function in Angular for form validation within reactive forms. Since you've finished the recipe, refer to the next section to see how this works.

How it works…

Angular provides an effortless way to create `async` validator functions, and they're handy too. We use an `async` validator function when there needs to be a validation that can be time-consuming and we don't want to block the main thread, or when we rely on a backend service for validation. For example, in this recipe, we started by creating the validator method named `versionValidator` inside a new service, called `VersionService`. Notice that we are using a couple of RxJS operators in the method, including `timer`, `switchMap`, and `map`. We use the `timer` operator to debounce and wait for the user to stop typing for 500 milliseconds. We then use the `switchMap` operator combined with the `validateVersion` method to make an HTTP call to the backend, validating the version. The benefit of `switchMap` here is that it cancels out existing HTTP calls, if any. Then, we use the `map` operator to remove the error from the HTTP call. If there's no error, we return an empty object from the map function's callback. If we have an error, we set the `[error]` as `true` in the `errors` object, as shown in *step 4*.

After creating the validator function, we attached it to the form control for the version name by using the `FormControl.setAsyncValidators` method in the `VersionControlComponent` class. We then used the validation errors, named `pattern` and `oldVersion`, in the template to show the relevant error message on the UI.

See also

- `AsyncValidator` Angular docs: `https://angular.io/api/forms/AsyncValidator#provide-a-custom-async-validator-directive`

Implementing complex forms with reactive FormArray

This is definitely one of the most requested recipes from the readers of the first edition of *Angular Cookbook*. In this recipe, we'll work with reactive Forms and specifically, the `FormArray` class from reactive forms. We're going to implement a complex form, which has a list of projects that can be submitted with reactive forms. The user will be able to add as many projects as they want and will be able to remove the ones they don't want.

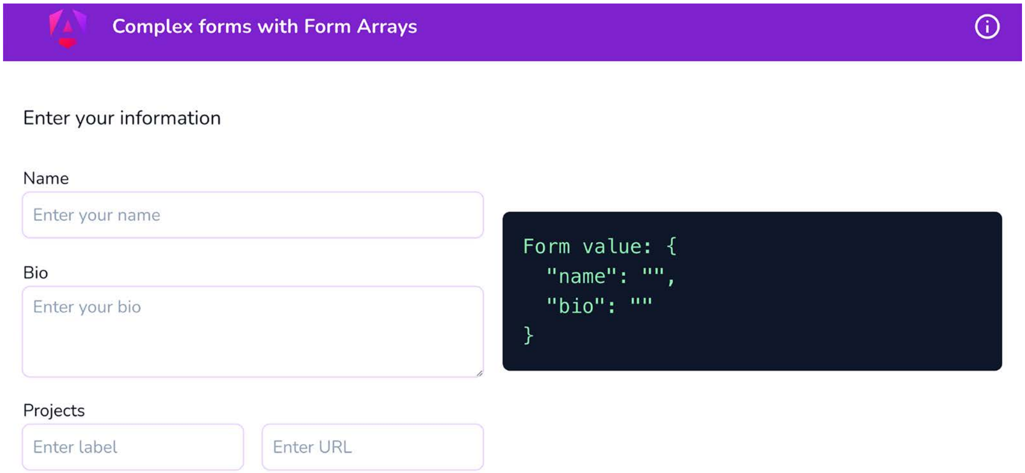
Getting ready

The app that we are going to work with resides in `start/apps/chapter08/ng-form-arrays` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-form-arrays
```

This should open the app in a new browser tab, and you should see the following:



```
Form value: {
  "name": "",
  "bio": ""
}
```

Figure 8.15: reactive FormArray app running on http://localhost:4200

Now that we have the app running locally, let's see the steps involved in this recipe in the next section.

How to do it...

We have an app that already implements a Reactive form. However, we can only add one project so far. We will use reactive `FormArray` to let the user submit multiple projects, and we will use the Reactive `FormBuilder`, which is a nice little API from the `@angular/forms` package that makes it easy to create and maintain reactive forms in Angular. Let us begin:

1. First of all, let's use the Reactive `FormBuilder` to add a `FormArray` to the existing Reactive form in the `portfolio-form.component.ts` file, as follows:

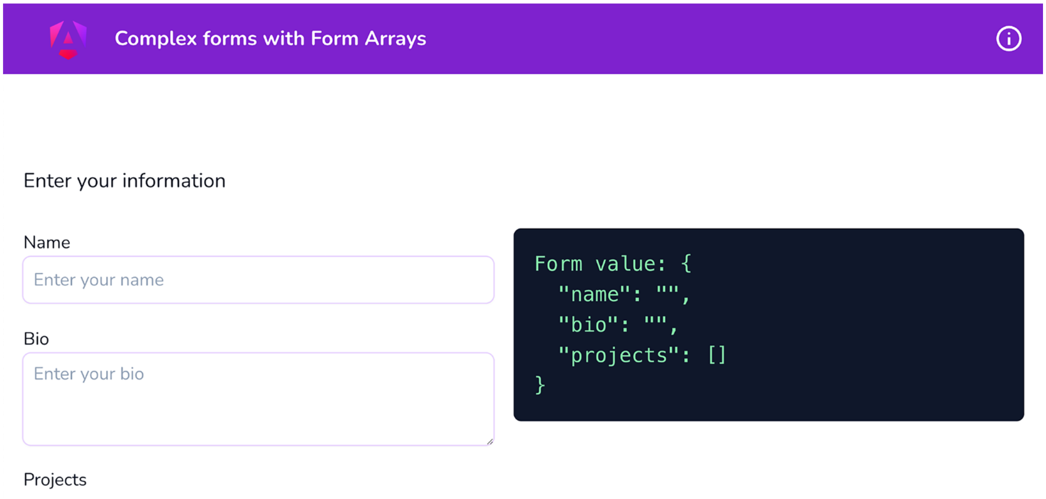
```
...
import { FormBuilder, ReactiveFormsModule, Validators, FormGroup,
FormControl } from '@angular/forms';
...
export class PortfolioFormComponent {
  fb = inject(FormBuilder);
  portfolioForm = this.fb.group({
    name: ['', Validators.required],
    bio: [''],
    projects: this.fb.array<
      FormGroup<{
        label: FormControl<string | null>;
        url: FormControl<string | null>;
      }>
    >([]),
  })

  get projectsFormArr() {
    return this.portfolioForm.controls.projects;
  }
  ...
}
```

2. Now that we have the `projects` `FormArray`, let's use that in the template so that we can show the form inputs for the projects, according to the form array. Let's modify the `portfolio-form.component.html` file, as follows:

```
<div class="flex gap-4 items-center flex-col md:flex-row">
  <form [formGroup]="portfolioForm" class="flex-1 w-full
    md:w-auto">
    <section>...</section>
    ...
    <section formArrayName="projects">
      <label class="text-sm">Projects</label>
      <fieldset class="flex gap-4 mb-4 justify-between"
```

```
        *ngFor="let projectControl of
        projectsFormArr.controls; trackBy: trackByFn"
        [formGroup]="projectControl">
        <input formControlName="label" type="text"
          class="form-control" placeholder="Enter label">
        <input formControlName="url" type="url"
          class="form-control" placeholder="Enter URL">
      </fieldset>
    </section>
  </form>
  ...
</div>
```

You'll notice that the object on the right in the app has a `projects` array now. However, we can see no inputs for the projects at the moment. That's because the form array is empty, as shown in the following figure:


Complex forms with Form Arrays
Enter your information
Name
Enter your name
Bio
Enter your bio
Projects

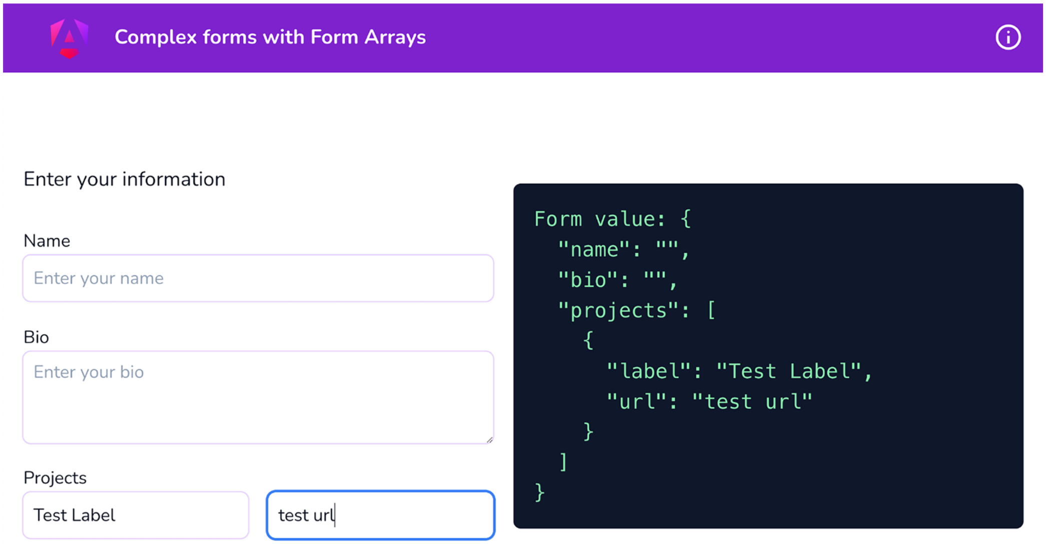
```
Form value: {
  "name": "",
  "bio": "",
  "projects": []
}
```

Figure 8.16: reactive FormArray app running on http://localhost:4200

3. We'll add one form group to the form array by default when the app starts. To do this, modify the `portfolio-form.component.ts` file, as follows:

```
import { Component, OnInit, inject } from '@angular/core';
...
export class PortfolioFormComponent implements OnInit {
  ...
  ngOnInit(): void {
    this.addNewProject();
    }

    addNewProject() {
      this.projectsFormArr.push(
        this.fb.group({
          label: ['', Validators.required],
          url: ['', Validators.required]
        })
      )
    }
    ...
  }
```

If you look at the app now, you should see one set of inputs for the projects, as shown in the following figure:


Complex forms with Form Arrays
Enter your information
Name
Enter your name
Bio
Enter your bio
Projects
Test Label
test url

```
Form value: {
  "name": "",
  "bio": "",
  "projects": [
    {
      "label": "Test Label",
      "url": "test url"
    }
  ]
}
```

Figure 8.17: One set of inputs for projects by default

4. Let's add a button for the user to be able to add more projects. We'll modify the `portfolio-form.component.html` file, as follows:

```
<section formArrayName="projects">
      <label class="text-sm">Projects</label>
      <fieldset [formGroup]="projectControl" class="flex
        gap-4 mb-4 justify-between" *ngFor="let
        projectControl of projectsFormArr.controls;
        let isLast = last; trackBy: trackByFn">
        ...
        <input formControlName="url" type="url" class="form-
          control" placeholder="Enter URL">
        <button type="button" [style.visibility]="isLast
          ? 'visible' : 'hidden'"
          (click)="addNewProject()">+</button>
      </fieldset>
   </section>
```

With the code above, you should see new project inputs appearing with the click of the + button.

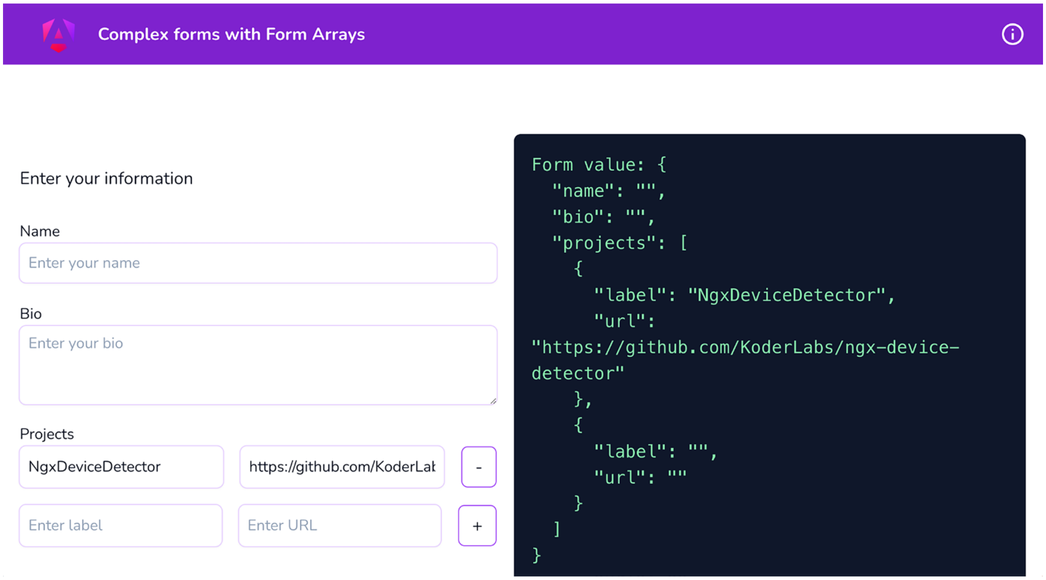
5. Let's add our final feature for this recipe, which is showing buttons to remove the project's input row. We'll first add a method in the `portfolio-form.component.ts` file that receives the index of the row to be removed as an argument:

```
...
export class PortfolioFormComponent implements OnInit {
  ...
  addNewProject() {...}

  removeProject(index: number) {
    this.projectsFormArr.removeAt(index);
  }
  ...
}
```

6. Now, we'll update the template to add the `Remove` button. Update the `portfolio-form.component.html` file, as follows:

```
<section formArrayName="projects">
      <label class="text-sm">Projects</label>
      <fieldset [formGroup]="projectControl" class="flex
        gap-4 mb-4 justify-between" *ngFor="let
        projectControl of projectsFormArr.controls; let
        isLast = last; let i = index; trackBy: trackByFn">
        <input formControlName="label" type="text"
          class="form-control" placeholder="Enter label">
        <input formControlName="url" type="url" class="form-
          control" placeholder="Enter URL">
        <button type="button" [hidden]="isLast"
          (click)="removeProject(i)">-</button>
        <button type="button" [style.visibility]="isLast ?
          'visible' : 'hidden'" [hidden]="!isLast"
          (click)="addNewProject()">+</button>
      </fieldset>
    </section>
```

If you now look at the app, you should be able to add or remove projects to the form as you like, as shown in the following figure:



Enter your information

Name

Enter your name

Bio

Enter your bio

Projects

NgxDeviceDetector | https://github.com/KoderLat | -

Enter label | Enter URL | +

```
Form value: {
  "name": "",
  "bio": "",
  "projects": [
    {
      "label": "NgxDeviceDetector",
      "url":
"https://github.com/KoderLabs/ngx-device-
detector"
    },
    {
      "label": "",
      "url": ""
    }
  ]
}
```

Figure 8.18: Final result with form arrays

Great! You now know how to work with reactive `FormArray`. Refer to the next section to understand how it all works.

How it works...

Angular's `FormArray` is an amazing tool built into the reactive Forms in Angular. It aggregates all values of each child `FormControl` or `FormGroup` into an array. The beauty of the `FormArray` is that if any one of the child `FormControls` is invalid, the whole array becomes invalid. In this recipe, we start by using the `FormBuilder` from the `ReactiveFormsModule` in the `PortfolioFormComponent` class. We initialize it with an empty `FormArray` and then push one `FormGroup` to it when the component mounts (using the `ngOnInit` method). This is so we can see one set of form inputs for the projects. Note that we use the `addNewProject` method for this. Also note that we use a `getter` function, named `projectsFormArr`, so that we can easily access the `FormArray` to be able to loop over it, add Form Groups to it, and remove the desired `FormGroups` from it when needed. We then loop over the `projectsFormArr.controls` array so that we can display the inputs via the HTML template, using the `*ngFor` directive.

Then, we modify the template to add the + button so that we can add more projects to the form array. Note the statement in the `*ngFor` has `let isLast = last`. The `*ngFor` directive provides us with the first and last variables automatically, so we can use them in our template. This statement creates a variable named `isLast` for us, and it assigns the value of the `last` variable provided by the `*ngFor` directive (which is a Boolean) to it. This is so we can show the + button only on the last row instead of on every row.

Finally, we add the `remove` button to each row except for the last one so we can remove the respective `FormGroup` from the `FormArray`. Note that here we also add `let i = index` to the template in the `*ngFor` directive. This is because our `removeProject` method in the TypeScript file expects the index of the `FormGroup` in the `FormArray` to be passed in it when it is called. Hence, we're able to get the index for each `FormGroup` from the `*ngFor` directive and can pass the index to the `removeProject` method. We use `[hidden]="isLast"` on each `remove` button so that it is hidden onthe last row only. This is because we show the + button on the last row and we need to have at least one empty row for the projects.

See also

- Angular `FormArray`: `https://angular.io/api/forms/FormArray`

Writing your own custom form control using ControlValueAccessor

Angular forms are great. While they support the default HTML tags like `input`, `textarea`, etc., sometimes, you might want to define your own components that take a value from the user. What if your components could work seamlessly with Angular forms, i.e., using `formControlName`, `ngModel`, and so on? That would be cool, right?

In this recipe, you'll learn how to create a component with your own custom form control using the `ControlValueAccessor` API, allowing you to use your component with both template-driven forms and reactive forms.

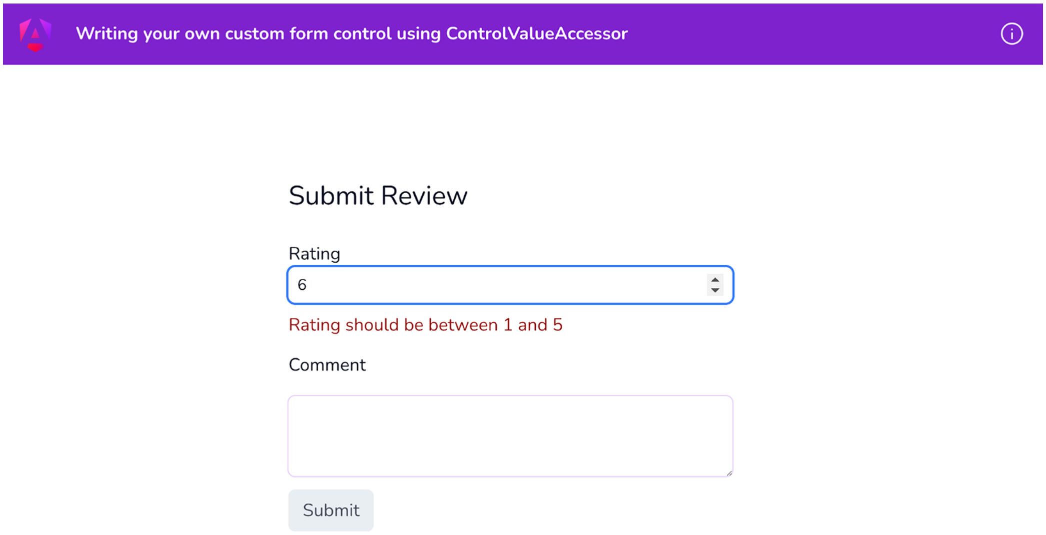
Getting ready

The app that we are going to work with resides in `start/apps/chapter08/ng-form-cva` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-form-cva
```

This should open the app in a new browser tab, and you should see the following:



Writing your own custom form control using ControlValueAccessor

Submit Review

Rating

6

Rating should be between 1 and 5

Comment

Figure 8.19: Custom form control app running on http://localhost:4200

Now that we have the app running locally, let's see the steps involved in this recipe in the next section.

How to do it...

We have a simple Angular app. It has two inputs and a **Submit** button. The inputs are for a review, and they ask the user to provide a value for the rating of this imaginary item and any comments the user wants to provide. We'll convert the rating input into a custom `FormControl` using the `ControlValueAccessor` API. Let's get started:

1. Let's create a component for our custom `FormControl` Open the terminal in the project root and run the following command and select the `@nx/angular:component schematics` and "As provided" when asked:

```
cd start && nx g c rating --directory apps/chapter08/ng-form-cva/src/app/
components/rating
```

2. We'll now create the `star` UI for the `rating` component. Modify the `rating.component.html` file, as follows:

```
<div class="rating">
  <div class="rating__star" [ngClass]="{
      'rating__star--active':
        (!isMouseOver && value >= star) ||
        (isMouseOver && hoveredRating >= star),
        '!cursor-default': disabled
    }" (mouseenter)="onRatingMouseEnter(star)"
         (mouseleave)="onRatingMouseLeave()"
         (click)="selectRating(star)"
    *ngFor="let star of [1, 2, 3, 4, 5]; let i = index">
    <span class="material-symbols-outlined">
      star
    </span>
  </div>
</div>
```

3. Add the styles for the `rating` component to the `rating.component.scss` file as follows:

```
.rating {
  display: flex;
  margin-bottom: 10px;
  &__star {
    cursor: pointer;
    color: grey;
    padding: 0 6px;
    &:first-child {
      padding-left: 0;
    }
    &:last-child {
      padding-right: 0;
    }
    &--active {
      color: orange;
    }
  }
}
```

4. We also need to modify the `RatingComponent` class to introduce the necessary methods and properties. Let's modify the `rating.component.ts` file, as follows:

```
/* eslint-disable @typescript-eslint/no-empty-function */
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';

@Component({
  ...
  standalone: true,
  imports: [CommonModule]
})
export class RatingComponent {
  value = 2;
  hoveredRating = 2;
  isMouseOver = false;

  onRatingMouseEnter(rating: number) {
    this.hoveredRating = rating;
    this.isMouseOver = true;
  }
  onRatingMouseLeave() {
    this.hoveredRating = 0;
    this.isMouseOver = false;
  }

  selectRating(rating: number) {
    this.value = rating;
  }

}
```

5. Let's import the `RatingComponent` class in the `HomeComponent` class because `RatingComponent` is a `standalone` component. Update the `home.component.ts` file, as follows:

```
...
import { RatingComponent } from '../components/rating/rating.component';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss'],
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule,
    RatingComponent]
})
export class HomeComponent {
  ...
}
```

6. Now, we need to use this `rating` component instead of the input that we already have in the `home.component.html` file. Modify the file as follows:

```
<div class="home">
  <div class="review-container">
    ...
    <form class="input-container" [formGroup]="reviewForm"
      (ngSubmit)="submitReview(reviewForm)">
      <div class="mb-3">
        <label for="ratingInput" class="form-
          label">Rating</label>
        <app-rating formControlName="rating"></app-rating>
      </div>
      <div class="mb-3">
        ...
      </div>
      <button id="submitBtn" [disabled]="reviewForm.
        invalid" class="btn btn-dark" type="submit">
        Submit</button>
    </form>
  </div>
</div>
```

If you refresh the app now and hover over the stars, you can see the color change. The selected rating is also highlighted as follows:

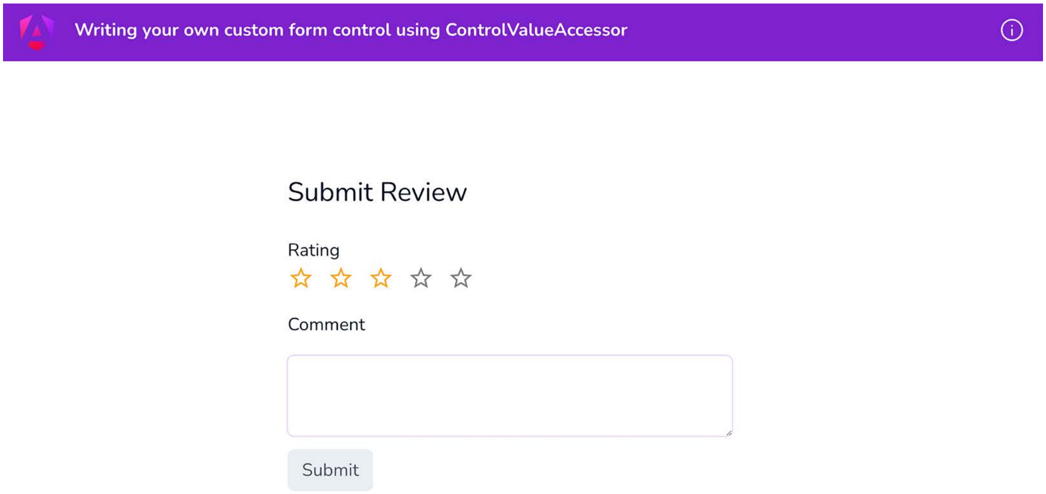

Writing your own custom form control using ControlValueAccessor
Submit Review
Rating
☆ ☆ ☆ ☆ ☆
Comment
Submit

Figure 8.20: Rating component with hovered stars

7. Let's now implement the `ControlValueAccessor` interface for our `rating` component. It requires a couple of methods to be implemented. Modify the `rating.component.ts` file, as follows:

```
/* eslint-disable @typescript-eslint/no-empty-function */
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { ControlValueAccessor } from '@angular/forms';
@Component({...})
export class RatingComponent implements ControlValueAccessor {
  ...
  isMouseOver = false;
  onChange: any = () => { };
  onTouched: any = () => { };
  ...
  registerOnChange(fn: any){
    this.onChange = fn;
  }
  registerOnTouched(fn: any) {
    this.onTouched = fn;
  }
}
```

8. We'll now add the required methods to disable the input when required and to set the value of the form control – in other words, the `setDisabledState` and `writeValue` methods. We'll also add the `disabled` property to our `RatingComponent` class, as follows:

```
...
import { Component, Input } from '@angular/core';
...
export class RatingComponent implements ControlValueAccessor {
  ...
  isMouseOver = false;
  disabled = false;
  ...
  setDisabledState(isDisabled: boolean): void {
    this.disabled = isDisabled;
  }
  writeValue(value: number) {
    this.value = value;
  }
}
```

9. We need to use the `disabled` property to prevent any UI changes when it is `true`. The value of the `value` variable should not be updated either. Modify the `rating.component.ts` file to do so, as follows:

```
...
@Component({...})
export class RatingComponent implements OnInit, ControlValueAccessor {
  ...
  isMouseOver = false;
  disabled = true;
  ...
  onRatingMouseEnter(rating: number) {
    if (this.disabled) return;
    this.hoveredRating = rating;
    this.isMouseOver = true;
  }
  ...
  selectRating(rating: number) {
    if (this.disabled) return;
    this.value = rating;
  }
  ...
}
```

With the above change, you'll notice that hovering over the stars doesn't do anything now because of the `disabled` property being set to `true`.

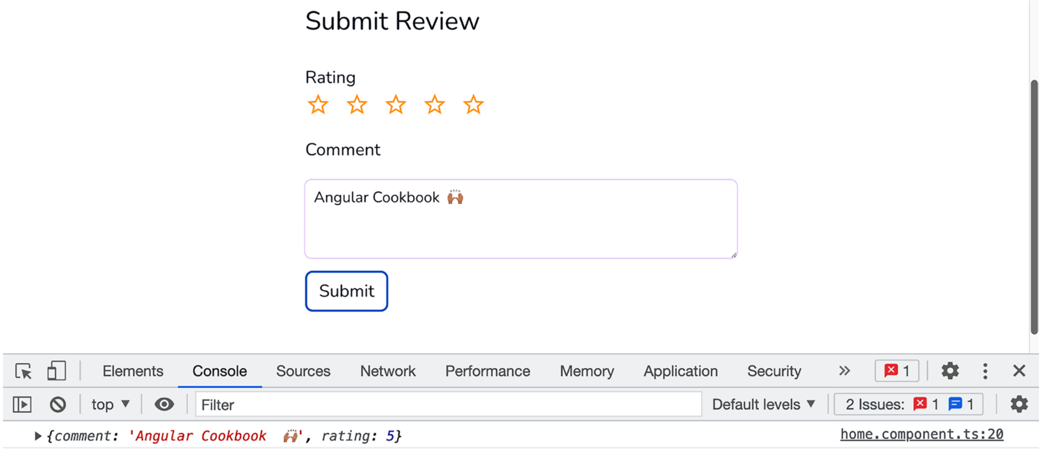
10. Let's make sure that we send the value of the `value` variable to `ControlValueAccessor` because that's what we want to access later. Also, let's set the `disabled` property back to `false`. Update the `selectRating` method in the `RatingComponent` class, as follows:

```
...
export class RatingComponent implements ControlValueAccessor {
  ...
  @Input() disabled = false;
  constructor() { }
  ...
  selectRating(rating: number) {
    if (this.disabled) return;
    this.value = rating;
    this.onTouched();
    this.onChange(rating);
  }
  ...
}
```

11. We need to tell Angular that our `RatingComponent` class has a `value` accessor; otherwise, using the `formControlName` attribute on the `<app-rating>` element will throw errors. Let's add an `NG_VALUE_ACCESSOR` provider to the `RatingComponent` class's decorator, as follows:

```
import { Component, forwardRef, Input, OnInit } from '@angular/core';
import { ControlValueAccessor, NG_VALUE_ACCESSOR } from '@angular/forms';
@Component({
  ...
  imports: [CommonModule],
  providers: [{
    provide: NG_VALUE_ACCESSOR,
    useExisting: forwardRef(() => RatingComponent),
    multi: true
  }]
})
...
```

If you refresh the app now, select a rating, and hit the **Submit** button, you should see the values being logged, as follows:



Submit Review

Rating ☆ ☆ ☆ ☆ ☆

Comment: Angular Cookbook 🙌

```
▸{comment: 'Angular Cookbook 🙌', rating: 5}                home.component.ts:20
```

Figure 8.21: Form value logged using the custom form control

Voilà! You just learned how to create a custom form control using `ControlValueAccessor`. Refer to the next section to understand how it works.

How it works...

We started the recipe by creating a component we can use to provide a rating for the review we submit. We started by adding the template and the styles for the `rating` component. Note that we are using an `[ngClass]` directive on each of the `star` elements to add the `rating__star--active` class conditionally. Let's discuss each condition now:

- `(isMouseOver && hoveredRating >= star)`: This condition relies on the `isMouseOver` and `hoveredRating` variables. The `isMouseOver` variable becomes `true` as soon as we mouse over any star and is turned back to `false` when we move away from the star. This means that it is only `true` when we're hovering over a star. `hoveredRating` tells us which star we're hovering over at any moment and is assigned the star's value – in other words, a value from `1` to `5`. So, this condition is only `true` when we're doing a mouseover, and the hovered star's rating is greater than the value of the current star. So, if we're hovering over the fourth star, all the stars from value `1` to `4` will be highlighted, as they'll have the `rating__star--active` class conditionally assigned to them.
- `(!isMouseOver && value >= star)`: This condition relies on the `isMouseOver` variable that we discussed previously and the `value` variable. The `value` variable holds the value of the selected rating, which is updated when we click on a star. So, this condition is applied when we're not doing a mouseover, and we have the value of the `value` variable greater than the current star. This is especially beneficial when you have a greater value assigned to the `value` variable and try to hover over a star with a lesser value, in which case, all the stars with values greater than the hovered star will not be highlighted.

Then, we used three events on each star, `mouseenter`, `mouseleave`, and `click`, and then used our `onRatingMouseEnter`, `onRatingMouseLeave`, and `selectRating` methods, respectively, for these events. All of this was designed to ensure that the entire UI is fluent and provides a good user experience. We then implemented the `ControlValueAccessor` interface for our `rating` component. When we do that, we need to define the `onChange` and `onTouched` methods as empty methods, which we did as follows:

```
onChange: any = () => { };
onTouched: any = () => { };
```

Then, we used the `registerOnChange` and `registerOnTouched` methods from `ControlValueAccessor` to assign our methods, as follows:

```
registerOnChange(fn: any){
  this.onChange = fn;
}
registerOnTouched(fn: any) {
  this.onTouched = fn;
}
```

We registered these methods because whenever we make a change in our component and want to let `ControlValueAccessor` know that the value has changed, we need to call the `onChange` method ourselves. We do that in the `selectRating` method as follows, which makes sure that when we select a rating, we set the form control's value to the value of the selected rating:

```
selectRating(rating: number) {
  if (this.disabled) return;
  this.value = rating;
  this.onChange(rating);
}
```

The other way around is when we need to know when the form control's value is changed from outside the component. In this case, we need to assign the updated value to the `value` variable. We do that in the `writeValue` method from the `ControlValueAccessor` interface, as follows:

```
writeValue(value: number) {
  this.value = value;
}
```

What if we don't want the user to provide a value for the rating? In other words, we want the rating form control to be disabled. For this, we use the `setDisabledState` method from the `ControlValueAccessor` interface, so whenever the form control's `disabled` state is changed, we set the `disabled` property to reflect the changes.

Finally, we wanted Angular to know that this `RatingComponent` class has a value accessor. This is so that we can use the reactive forms API – specifically, the `formControlName` attribute with the `<app-rating>` selector – and use it as a form control. To do that, we provide our `RatingComponent` class as a provider to its `@Component` definition decorator, using the `NG_VALUE_ACCESSOR` injection token, as follows:

```
@Component({
  ...
  providers: [{
    provide: NG_VALUE_ACCESSOR,
    useExisting: forwardRef(() => RatingComponent),
    multi: true
  }]
})
```

Note that we're using the `useExisting` property with the `forwardRef` method providing our `RatingComponent` class in it. We need to provide `multi: true` because Angular itself registers some value accessors using the `NG_VALUE_ACCESSOR` injection token, and there may also be third-party form controls.

Once we've set everything up, we use `formControlName` on our `rating` component in the `home.component.html` file, as follows:

```
<app-rating formControlName="rating"></app-rating>
```

See also

- Custom form control in Angular by thoughtram: `https://blog.thoughtram.io/angular/2016/07/27/custom-form-controls-in-angular-2.html`
- `ControlValueAccessor` docs: `https://angular.io/api/forms/ControlValueAccessor`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

9

Angular and the Angular CDK

Angular has an amazing ecosystem of tools and libraries, including Angular Material, the **Angular command-line interface** (**Angular CLI**), and the beloved **Angular Component Dev Kit** (**Angular CDK**). I call it "beloved" because if you are to implement your own custom interactions and behaviors in Angular apps without having to rely on an entire set of libraries, the Angular CDK is going to be your best friend. In this chapter, you'll learn what an amazing combination of Angular and the Angular CDK is. You'll learn about some neat components built into the CDK and will also use some CDK **APIs** to create amazing and optimized content.

Here are the recipes we're going to cover in this chapter:

- Using virtual scroll for huge lists
- Keyboard navigation for lists
- Pointy little popovers with the Overlay API
- Input coercion utilities from the Angular CDK
- Using the CDK drag and drop API to move items from one list to another
- Creating a multi-step game with the CDK stepper API
- Accessible Listbox interactions using CDK listbox directives
- Working with nested menus using the Angular CDK menu API

Technical requirements

For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter09`.

Using virtual scroll for huge lists

There might be certain scenarios in your application where you might have to show a huge set of items. This could be from either your backend API or the browser's local storage. In either case, rendering a lot of items at once causes performance issues because the **Document Object Model** (**DOM**) struggles, and also because of the fact that the JS thread gets blocked, and the page becomes unresponsive. In this recipe, we'll render a list of 270,000 users and use the `virtual scroll` functionality from the Angular CDK to improve the rendering performance.

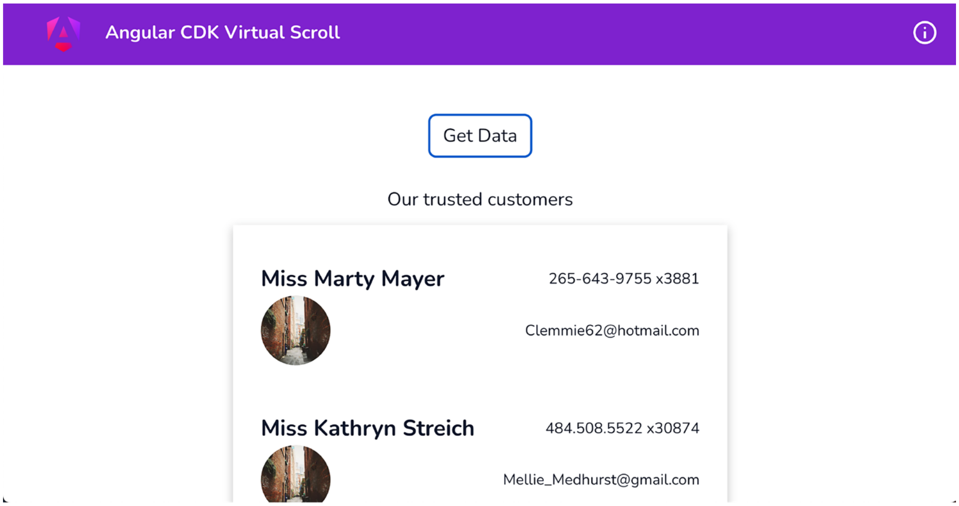
Getting ready

The app that we are going to work with resides in `start/apps/chapter09/ng-cdk-virtual-scroll` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-cdk-virtual-scroll
```

This should open the app in a new browser tab. When you click the **Get Data** button, you should see the following:



Miss Marty Mayer
265-643-9755 x3881
Clemmie62@hotmail.com

Miss Kathryn Streich
484.508.5522 x30874
Mellie_Medhurst@gmail.com

Figure 9.1: The ng-cdk-virtual-scroll app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it…

We have a pretty simple Angular app, but with a lot of data. Right now, it shows a loader (button) for about a couple of seconds and then is supposed to show the data. However, you'll notice that right after the **Tap me** button disappears, the loader keeps showing, the button is unresponsive, and we see a blank screen, as follows:

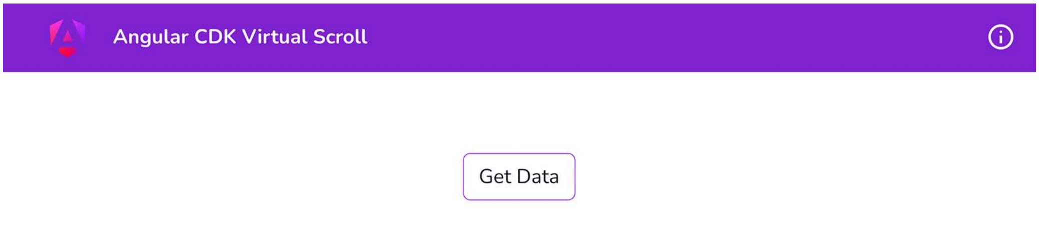


Figure 9.2: App stuck with a blank screen while rendering list items

In fact, our entire application becomes unresponsive. If you scroll through—or even hover over—the items, you'll see that the hover animation on the list items is not smooth and is a bit laggy. Let's see the steps to use the Angular CDK virtual scroll to improve the rendering performance, as follows:

1. We already have the Angular CDK installed in the workspace. However, if you needed to install it, you would run the following command from the project's root:

```
cd start && npm install --save @angular/cdk
```

2. You may have to restart your Angular server if you reinstall the package, so rerun the `npm run serve ng-cdk-virtual-scroll` command again.
3. Add the `ScrollingModule` class from the `@angular/cdk` package into the `users-list.component.ts` file, as follows:

```
...
import { ScrollingModule } from '@angular/cdk/scrolling';
@Component({
  ...,
  imports: [CommonModule, ScrollingModule]
})
export class UsersListComponent {
  @Input() listItems: AppUserCard[] = [];
}
```

We now must implement the virtual scroll, modify the `users-list-item.component.html` file to use the `*cdkVirtualFor` directive instead of the `*ngFor directive`, and wrap the `<li>` element inside a `<cdk-virtual-scroll-viewport>` element, as follows:

```
<h4 class="heading">Our trusted customers</h4>
<ul>
  <cdk-virtual-scroll-viewport class="list list-group p-2"
    [itemSize]="120">
    <li class="list__item list-group-item my-2 rounded-md p -4"
      *cdkVirtualFor="let item of listItems">
      ...
      </li>
    </cdk-virtual-scroll-viewport>
  </ul>
```

Notice that we have moved the CSS classes `"list list-group p-2"` from the `<ul>` element to the `<cdk-virtual-scroll-viewport>` element.

Boom! Within a few steps, and by using the Angular CDK virtual scroll, we were able to fix a huge performance rendering issue within our Angular app. See the next section to understand how it works.

How it works...

The Angular CDK provides the Scrolling APIs, which include the `*cdkVirtualFor` directive and the `<cdk-virtual-scroll-viewport>` element. It is necessary to have `<cdk-virtual-scroll-viewport>` wrapping the element that has the `*cdkVirtualFor` directive being applied to it. Notice that we have an attribute on the `cdk-virtual-scroll-viewport` element named `[itemSize]`, having its value set to `"120"`. The reason for this is that each list item has a height of approximately 120 pixels, as shown in the following screenshot:

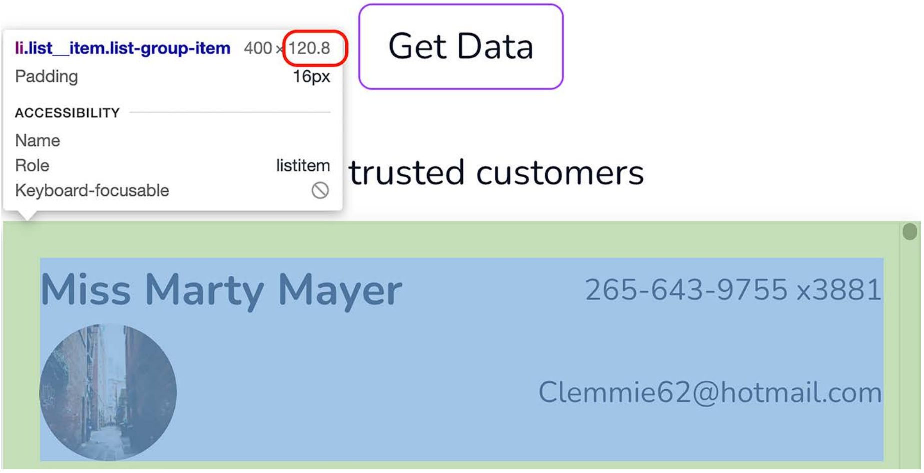
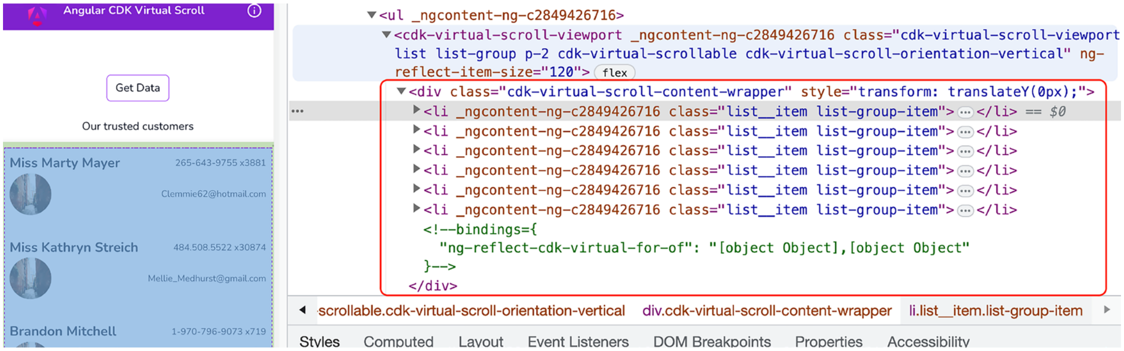

li.list__item.list-group-item 400 × 120.8
Padding 16px
ACCESSIBILITY
Name
Role listitem
Keyboard-focusable
Get Data
trusted customers
Miss Marty Mayer
265-643-9755 x3881
Clemmie62@hotmail.com

Figure 9.3: Each list item has a height of approximately 120 pixels

But how does it improve the rendering performance? Glad you asked! In the original code for this recipe, when we loaded the 270,000 users, it would create a separate `<li>` element with the `class="list__item list-group-item"` attribute for each user, thus creating 270,000 DOM elements all being rendered at once. With the virtual scroll in place, the CDK only creates a few `<li>` elements, renders them, and just replaces the content of those few `<li>` elements as we scroll through items.

For our example, it creates 6 `<li>` elements, as shown in the following screenshot:


Angular CDK Virtual Scroll
Get Data
Our trusted customers
Miss Marty Mayer
265-643-9755 x3881
Clemmie62@hotmail.com
Miss Kathryn Streich
484.508.5522 x30874
Mellie_Medhurst@gmail.com
Brandon Mitchell
1-970-796-9073 x719
scrollable.cdk-virtual-scroll-orientation-vertical
div.cdk-virtual-scroll-content-wrapper
li.list__item.list-group-item
Styles
Computed
Layout
Event Listeners
DOM Breakpoints
Properties
Accessibility

```
▼<ul _ngcontent-ng-c2849426716>
  ▼<cdk-virtual-scroll-viewport _ngcontent-ng-c2849426716 class="cdk-virtual-scroll-viewport
   list list-group p-2 cdk-virtual-scrollable cdk-virtual-scroll-orientation-vertical" ng-
   reflect-item-size="120"> flex
    ▼<div class="cdk-virtual-scroll-content-wrapper" style="transform: translateY(0px);">
…     ▶<li _ngcontent-ng-c2849426716 class="list__item list-group-item">…</li> == $0
      ▶<li _ngcontent-ng-c2849426716 class="list__item list-group-item">…</li>
      ▶<li _ngcontent-ng-c2849426716 class="list__item list-group-item">…</li>
      ▶<li _ngcontent-ng-c2849426716 class="list__item list-group-item">…</li>
      ▶<li _ngcontent-ng-c2849426716 class="list__item list-group-item">…</li>
      ▶<li _ngcontent-ng-c2849426716 class="list__item list-group-item">…</li>
      <!--bindings={
        "ng-reflect-cdk-virtual-for-of": "[object Object],[object Object"
      }-->
    </div>
```

Figure 9.4: Showing only a few <div> elements rendered on DOM due to virtual scroll

Since we only have a few elements rendered on the DOM, we don't have performance issues anymore, and the hover animation also seems super-smooth now.

> When implementing virtual scroll in your own applications, make sure that you set a specific height to the `<cdk-virtual-scroll viewport>` element, and also set the `[itemSize]` attribute equal to the expected list-item height in pixels, otherwise the list won't show.

See also

- Angular CDK scrolling examples: `https://material.angular.io/cdk/scrolling/examples`

Keyboard navigation for lists

Accessibility is one of the most important aspects of building apps with a great user experience. The apps should not only be fast and performant but also accessible. While there are a lot of things to consider when it comes to accessibility, in this recipe, we're going to make lists and list items more accessible by providing keyboard navigation for the items. With the Angular CDK, it is super simple. We're going to use the `ListKeyManager` service from the Angular CDK to implement keyboard navigation for the users list in our target application.

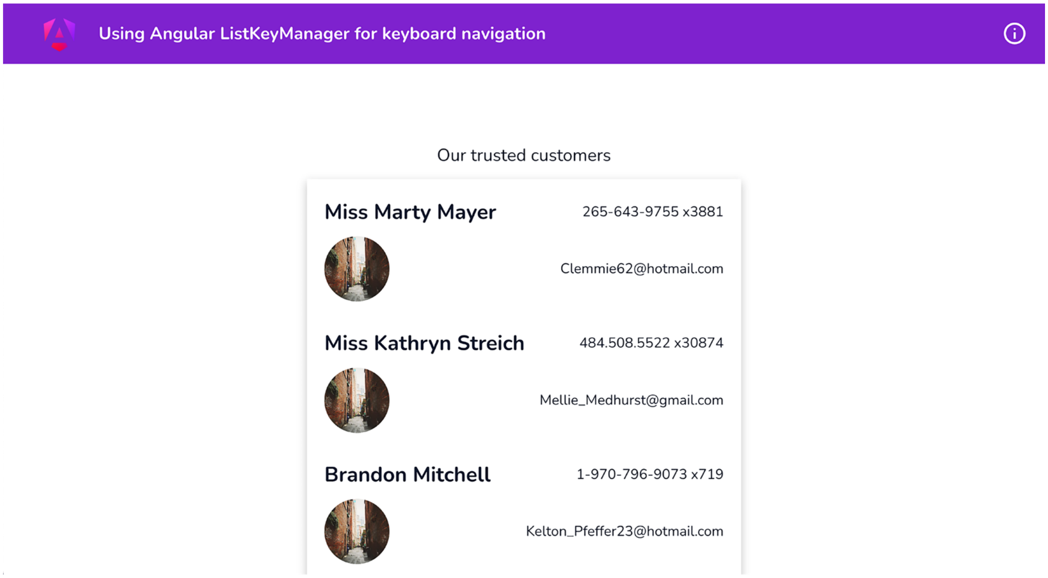
Getting ready

The app that we are going to work with resides in `start/apps/chapter09/ng-cdk-lkm` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-cdk-lkm
```

This should open the app in a new browser tab, and you should see the following:



Our trusted customers

Miss Marty Mayer
265-643-9755 x3881
Clemmie62@hotmail.com

Miss Kathryn Streich
484.508.5522 x30874
Mellie_Medhurst@gmail.com

Brandon Mitchell
1-970-796-9073 x719
Kelton_Pfeffer23@hotmail.com

Figure 9.5: The ng-cdk-lkm app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

We have an app that already has some of the goodness of the Angular CDK—that is, it has virtual scroll implemented from the previous recipe. We'll now start making changes to the app to implement keyboard navigation, as follows:

1. First, we'll implement the `FocusableOption` interface and some accessibility factors to our `UsersListItemComponent` class, as follows:

```
import { Component, HostBinding, Input, ViewEncapsulation } from '@
angular/core';
...
import { FocusableOption } from '@angular/cdk/a11y';

@Component({
  ...,
  styleUrls: ['./users-list-item.component.scss'],
  encapsulation: ViewEncapsulation.None,
})
export class UsersListItemComponent implements FocusableOption {
  @Input() item!: Partial<AppUserCard>;
  @HostBinding('tabIndex')
  tabIndex = -1;
  @HostBinding('role')
  role = 'list-item';
  focus() {}
}
```

2. We now need to implement what happens in the `focus` method. We'll use the `ElementRef` service to get the `nativeElement` and will set focus on the `nativeElement`, as follows:

```
import { Component, ElementRef, HostBinding, Input, ViewEncapsulation,
inject } from '@angular/core';
...
export class UsersListItemComponent implements FocusableOption {
  ...
  role = 'list-item';
  el = inject(ElementRef);

  focus() {
    this.el.nativeElement.focus();
  }
}
```

3. We will now add some accessibility to the list itself. Update the file `users-list.component.ts` as follows:

```
import { Component, HostBinding, Input } from '@angular/core';
...
export class UsersListComponent {
  @HostBinding('role')
  role = 'list';
  @Input() listItems: AppUserCard[] = [];
}
```

4. We now need to implement the `FocusKeyManager` class in our `UsersListComponent` class. We'll have to query our list items in the component to create an instance of the `FocusKeyManager` class. Update the `users-list.component.ts` file, as follows:

```
import { Component, HostBinding, Input, QueryList, ViewChildren } from '@
angular/core';
...
import { FocusKeyManager } from '@angular/cdk/a11y';
...
export class UsersListComponent {
  @HostBinding('role')
  role = 'list';
  @Input() listItems: AppUserCard[] = [];
  @ViewChildren(UsersListItemComponent)
  listItemsElements!: QueryList<UsersListItemComponent>;
  private listKeyManager!: FocusKeyManager<UsersListItemComponent>;
}
```

5. We'll now initiate the `FocusKeyManager` instance in the `AfterViewInit` hook in the same file. Update the `users-list.component.ts` file as follows:

```
import { AfterViewInit, Component, HostBinding, Input, QueryList,
ViewChildren } from '@angular/core';
      ...
export class UsersListComponent implements AfterViewInit {
  ...
  ngAfterViewInit() {
    this.listKeyManager = new FocusKeyManager(
      this.listItemsElements
    );
  }
}
```

6. Finally, we need to listen to the keyboard events. For this, you could either use a `keydown` event or a `window:keydown` event. For simplicity of the recipe, we'll go with the `window:keydown` event as pressing any key will bubble up the event to the `window` object. Update the file as follows:

```
import { AfterViewInit, Component, HostBinding, HostListener, Input,
QueryList, ViewChildren } from '@angular/core';
...
export class UsersListComponent implements AfterViewInit {
  ...
  @HostListener('window:keydown', ['$event'])
  onKeydown(event: KeyboardEvent) {
    this.listKeyManager.onKeydown(event);
  }
  ...
}
```

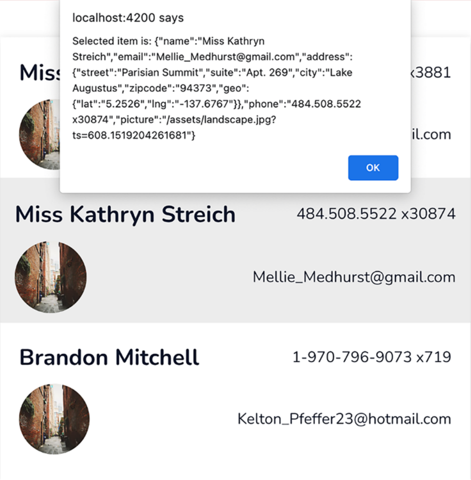
   If you go to the app, and press the down arrow key, you should see the first item being focused on the list and you can navigate to next and previous items and focusing them using the arrow keys.

7. To add a bit more fun to the recipe, let's also implement an event listener for when the user presses the *Enter* key on a particular item. We will just show an alert with the item's details. Let's use a `HostListener` decorator in the `users-list-item.component.ts` file as follows:

```
import { Component, ElementRef, HostBinding, HostListener, Input,
ViewEncapsulation, inject } from '@angular/core';
```

```
...
export class UsersListItemComponent implements FocusableOption {
  ...
  role = 'list-item';
  @HostListener('keyup', ['$event'])
  onEnter(ev: KeyboardEvent) {
    if (ev.code === 'Enter') {
      alert('Selected item is: ' +
        JSON.stringify(this.item));
    }
  }
  ...
}
```

With the above change, you should be able to see the selected item as shown in the following figure:



```
localhost:4200 says

Selected item is: {"name":"Miss Kathryn
Streich","email":"Mellie_Medhurst@gmail.com","address":
{"street":"Parisian Summit","suite":"Apt. 269","city":"Lake
Augustus","zipcode":"94373","geo":
{"lat":"5.2526","lng":"-137.6767"}},"phone":"484.508.5522
x30874","picture":"/assets/landscape.jpg?
ts=608.1519204261681"}
```

OK
Miss
x3881
il.com
Miss Kathryn Streich
484.508.5522 x30874
Mellie_Medhurst@gmail.com
Brandon Mitchell
1-970-796-9073 x719
Kelton_Pfeffer23@hotmail.com

Figure 9.6: Selected item JSON being shown in an alert

Awesomesauce! You have just learned how to implement keyboard navigation using the Angular CDK. See the next section to understand how it works.

How it works...

The Angular CDK provides the `ListKeyManager` class, which allows you to implement keyboard navigation. There are a bunch of techniques we can use with the `ListKeyManager` class, and for this recipe, we chose the `FocusKeyManager` class. To make it work for a list of items, we needed to do the following things:

1. We made sure that we had a component for each item in the list.

2. We used `ViewChildren` with `QueryList` in the list component to query all the list item components. `ViewChildren` is a property decorator that retrieves all the `UsersListItemComponent` elements from the DOM using `QueryList`, which gets updated automatically if an item is added, removed, or updated in the list.
3. We then created an instance of the `FocusKeyManager` class in the list component so we can initiate the list item component elements.
4. Added a keyboard listener to the list component and passed the event to the instance of the `FocusKeyManager` class.

When we defined the `listKeyManager` property in the `UsersListComponent` class, we also defined its type as well by specifying it as `FocusKeyManager<UsersListItemComponent>`. This made it easier to understand that our `FocusKeyManager` class is supposed to work with an array of `UsersListItemComponent` elements. Therefore, in the `ngAfterViewInit` method, we specified `this.listKeyManager = new FocusKeyManager(this.listItemsElements);`, which provided a queried list of `UsersListItemComponent` elements.

Finally, when we listened to the `window:keydown` event, we took the `keydown` event received in the handler and provided it to the instance of our `FocusKeyManager` class as `this.listKeyManager.onKeydown(event);`. This told our `FocusKeyManager` instance which key was pressed and what it must do.

Notice that our `UsersListItemComponent` class implements the `FocusableOption` interface, and it also has the `focus` method, which the `FocusKeyManager` class uses behind the scenes when we press the keyboard arrow-down or arrow-up keys.

Finally, we also used two properties with the `HostBinding` decorator. Let's analyze them one by one:

```
@HostBinding('tabIndex') and tabIndex = -1;
```

The `HostBinding` decorator is used here to bind a class property (`tabIndex`) to a host element's attribute (`tabIndex`). This effectively sets the `tabIndex` of the component's host element to `-1`, making it programmatically focusable but not reachable via sequential keyboard navigation.

```
@HostBinding('role') and role = 'list-item';
```

Similarly, the above binds the role property of the class to the role attribute of the component's host element. It assigns the ARIA role list item to the host element, providing a hint to assistive technologies about how they should interpret this element.

Now that you know how the recipe works, see the next section for some relevant links.

See also

- Angular CDK accessibility documentation: `https://material.angular.io/cdk/a11y/overview`
- `ViewChildren` official documentation: `https://angular.io/api/core/ViewChildren`

Pointy little popovers with the Overlay API

This is one of the advanced recipes in this book, especially for those of you who have already been working with Angular for a while. In this recipe, we'll not only create some **popovers** using the **CDK Overlay API**, but we'll also make them pointy, just like tooltips, and that's where the fun lies.

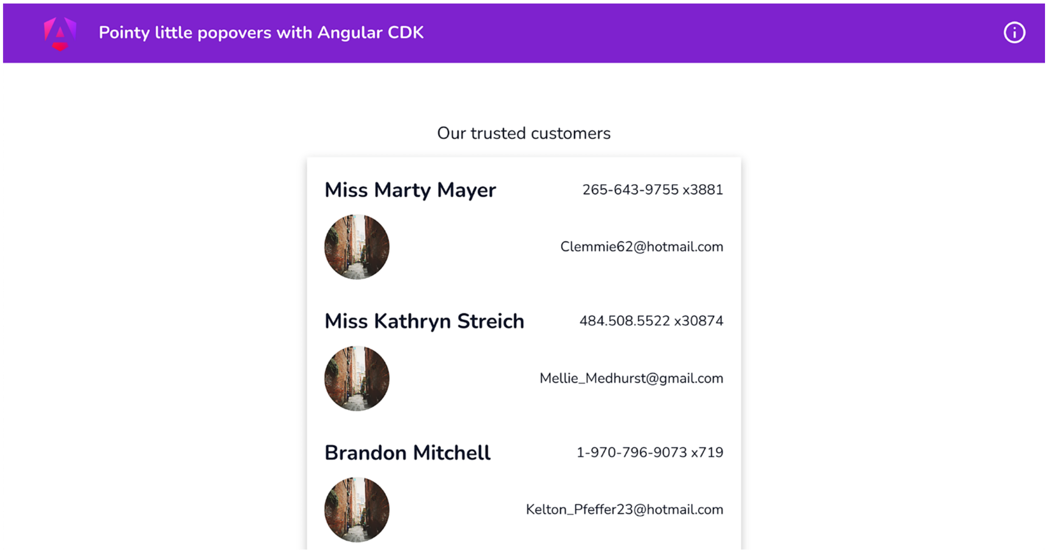
Getting ready

The app that we are going to work with resides in `start/apps/chapter09/ng-cdk-popover` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

    ```
    npm run serve ng-cdk-popover
    ```

    This should open the app in a new browser tab, and you should see the following:



Pointy little popovers with Angular CDK

Our trusted customers

Miss Marty Mayer 265-643-9755 x3881
Clemmie62@hotmail.com

Miss Kathryn Streich 484.508.5522 x30874
Mellie_Medhurst@gmail.com

Brandon Mitchell 1-970-796-9073 x719
Kelton_Pfeffer23@hotmail.com

Figure 9.7: The ng-cdk-popover app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it…

Our app has a list of users that we can scroll through on the page. We'll add a popover menu to each item so that a drop-down menu is shown with some actions. We already have the `@angular/cdk` package installed, so we don't need to worry about that. Let's start with the recipe, as follows:

1. We'll first add the Overlay's default styles so that when the overlay is displayed, it is positioned correctly. Open the `src/styles.scss` file and update it as per the following snippet:

    `https://ng-cookbook.com/s/cdk-pane-styles`

2. Now, we'll create variables to hold the overlay origin (for the origin of the positions of the opened overlay) and the actual relative position's settings. Open the `users-list.component.ts` file and update it, as follows:

```
...
import { CdkOverlayOrigin, ConnectedPosition, OverlayModule } from '@
angular/cdk/overlay';
...
@Component({
  ...,
  imports: [..., OverlayModule],
})
export class UsersListComponent {
  @Input() listItems: AppUserCard[] = [];
  popoverMenuOrigin!: CdkOverlayOrigin | null;
  menuPositions: ConnectedPosition[] = [
    { offsetY: 4, originX: 'end', originY: 'bottom',
      overlayX: 'end', overlayY: 'top' },
    { offsetY: -4, originX: 'end', originY: 'top', overlayX:
      'end', overlayY: 'bottom' },
  ];
     }
```

3. Now, open the `users-list.component.html` file and add the `cdkOverlayOrigin` directive to the `<app-users-list-item>` selector so that we can have each list item as an origin for the pop-up menu, as follows:

```
<h4 class="heading">Our trusted customers</h4>
<ul>
  <cdk-virtual-scroll-viewport class="list list-group p-2"
    [itemSize]="120">
    <app-users-list-item
      class="list__item"
      cdkOverlayOrigin #menuTrigger="cdkOverlayOrigin"
      *cdkVirtualFor="let item of listItems"
      [item]="item">
    </app-users-list-item>
  </cdk-virtual-scroll-viewport>
</ul>
```

4. We need to somehow pass the `#menuTrigger` variable from the template to assign its value to the `popoverMenuOrigin` property in the `UsersListComponent` class. To do so, create a method named `openMenu` in the `users-list.component.ts` file, as follows:

   ```
   ...
   export class UsersListComponent {
     ...
     openMenu($event: Event, itemTrigger: CdkOverlayOrigin) {
       if ($event) {
         $event.stopImmediatePropagation();
       }
       this.popoverMenuOrigin = itemTrigger;
     }
   }
   ```

5. We also need a property to show/hide the popover menu. Let's create it and set it to `true` in the `openMenu` method as well. Update the `users-list.component.ts` file, as follows:

   ```
   ...
   export class UsersListComponent{
     ...
     menuShown = false;
     ...
     openMenu($event: Event, itemTrigger: CdkOverlayOrigin) {
       if ($event) {
         $event.stopImmediatePropagation();
       }
       this.popoverMenuOrigin = itemTrigger;
       this.menuShown = true;
     }
     closeMenu() {
       this.popoverMenuOrigin = null;
       this.menuShown = false;
     }
     ...
   }
   ```

6. We'll now create an actual overlay. To do so, we'll create an `<ng-template>` element with the `cdkConnectedOverlay` directive. At the bottom of the `users-list.component.html` file, add the code (`<ng-template> element`) from the following link:

   `https://ng-cookbook.com/s/ng-popover-overlay`

7. We need to pass the `#menuTrigger` variable that we have on each list item to the `openMenu` method on a click of the list item. Update the file, as follows:

```
<h4 class="heading">Our trusted customers</h4>
<ul>
  <cdk-virtual-scroll-viewport class="list list-group p-2"
    [itemSize]="120">
    <app-users-list-item
      class="list__item"
      cdkOverlayOrigin #menuTrigger="cdkOverlayOrigin"
      *cdkVirtualFor="let item of listItems"
      (click)="openMenu($event, menuTrigger)"
      [class.list__item--active]="popoverMenuOrigin ===
        menuTrigger"
      [item]="item">
    </app-users-list-item>
  </cdk-virtual-scroll-viewport>
</ul>
...
```

If you refresh the app now and click on any of the list items, you should see a drop-down menu, as follows:

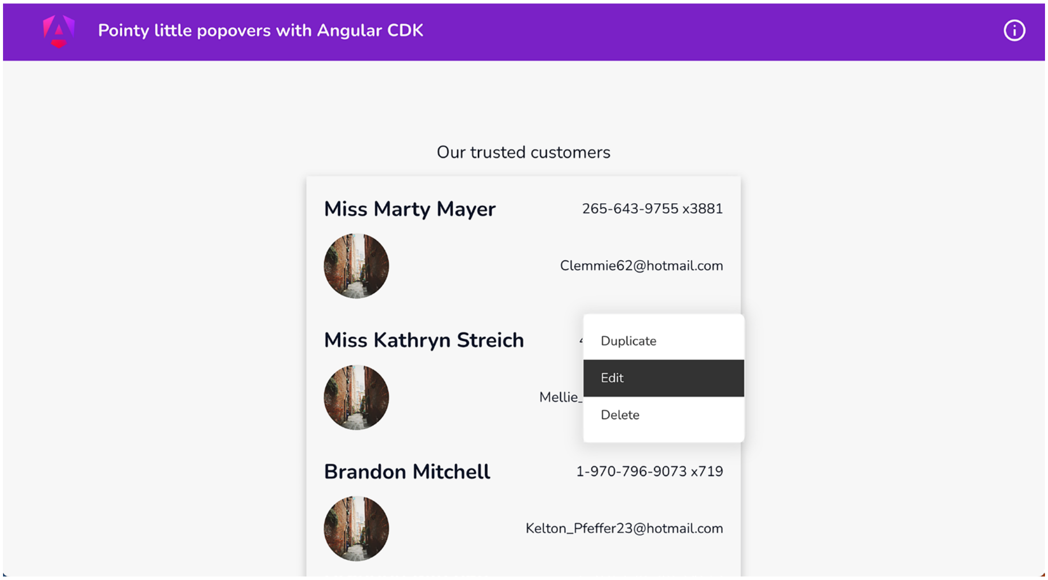

Pointy little popovers with Angular CDK
Our trusted customers
Miss Marty Mayer
265-643-9755 x3881
Clemmie62@hotmail.com
Miss Kathryn Streich
Mellie_
Duplicate
Edit
Delete
Brandon Mitchell
1-970-796-9073 x719
Kelton_Pfeffer23@hotmail.com

Figure 9.8: Working drop-down menu for each list item

8. We now have to implement the part where we show a pointy little arrow with the drop-down menu so that we can correlate the drop-down menu with the list item. First, add the following styles to the `menu-popover` class in the `src/styles.scss` file:

```
...
.menu-popover {
  ...
  &::before {
    top: -10px;
    border-width: 0px 10px 10px 10px;
    border-color: transparent transparent white
      transparent;
    position: absolute;
    content: '';
    right: 5%;
    border-style: solid;
  }
  &--up {
    transform: translateY(-20px);
    &::before {
      top: unset !important;
      transform: rotate(180deg);
      bottom: -10px;
    }
  }
  &__list {...}
}
```

You should now be able to see a pointy arrow on the top right of the drop-down menu, but if you try clicking the last item on the screen, you'll see that the drop-down menu opens upward but still shows the pointer at the top, as follows:

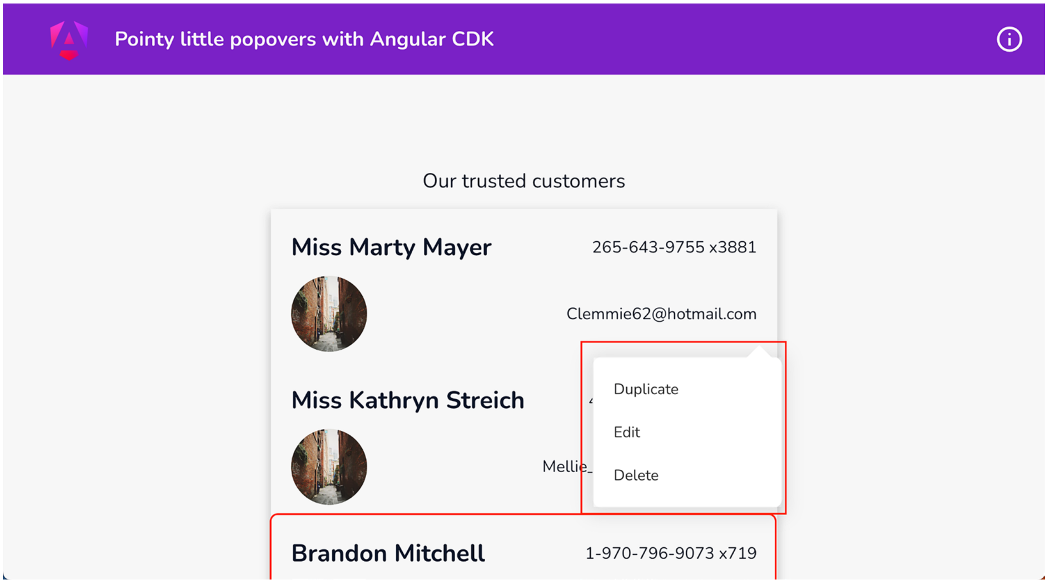

Pointy little popovers with Angular CDK
Our trusted customers
Miss Marty Mayer
265-643-9755 x3881
Clemmie62@hotmail.com
Miss Kathryn Streich
Mellie_
Duplicate
Edit
Delete
Brandon Mitchell
1-970-796-9073 x719

Figure 9.9: Drop-down arrow pointing at the wrong list item

9. To point to the actual origin of the popover/drop-down menu, we need to implement a custom directive that applies a custom class to the popover. Let's start by creating a directive, as follows:

```
cd start && nx g directive positional-popover-class --directory apps/
chapter09/ng-cdk-popover/src/app/directives
```

10. Update the code in the `popover-positional-class.directive.ts` generated file as per the following snippet:

    `https://ng-cookbook.com/s/popover-pc-directive`

11. Let's import the `PopoverPositionalClassDirective` class in the file `users-list.component.ts` so we can use it in the template later. Update the code as follows:

```
...
import { PopoverPositionalClassDirective } from '../../directives/
popover-positional-class.directive';

@Component({
  ...
  imports: [CommonModule, ScrollingModule, UsersListItemComponent,
OverlayModule, PopoverPositionalClassDirective]
})
...
```

12. Now, open the `users-list.component.html` file to apply our directive to the `<ng-template>` element. Update the file as follows:

```
...
<ng-template
  cdkConnectedOverlay
  ...
  (backdropClick)="closeMenu()"
  [cdkConnectedOverlayPositions]="menuPositions"
  appPopoverPositionalClass targetSelector=".menu-popover"
    inverseClass="menu-popover--up"
  [originY]="popoverMenuPosition.originY"
    (positionChange)="popoverPositionChanged($event)"
  cdkConnectedOverlayPanelClass="menu-popover"
  >
  <div class="menu-popover__list">
    ...
  </div>
</ng-template>
```

13. We now need to create a `popoverMenuPosition` property and a `popoverPositionChanged` method in our `users-list.component.ts` file to track which list item is for opening the menu (for positioning) and which is for when the position of the menu changes due to resizing of the window or a change in content. Update the file further as follows:

```
...
import { ChangeDetectorRef, Component, Input, inject } from '@angular/
core';
...
import { CdkOverlayOrigin, ConnectedOverlayPositionChange,
ConnectedPosition, OverlayModule } from '@angular/cdk/overlay';
@Component({...})
export class UsersListComponent {
  ...
   cdRef = inject(ChangeDetectorRef);

  popoverMenuPosition: Partial<ConnectedPosition> = {
    originY: undefined
  };

  popoverPositionChanged($event:
    ConnectedOverlayPositionChange) {
    if (this.popoverMenuPosition.originY !==
      $event.connectionPair.originY) {
      this.popoverMenuPosition.originY = $event.connectionPair.originY;
      }
    this.cdRef.detectChanges();
  }
  ...
}
```

And voilà! If you now refresh the page and tap each of the list items, you'll see the arrows point in the correct direction. See the following screenshot to view the popover arrow pointing downward for the last item, due to the popover being shown above the item:

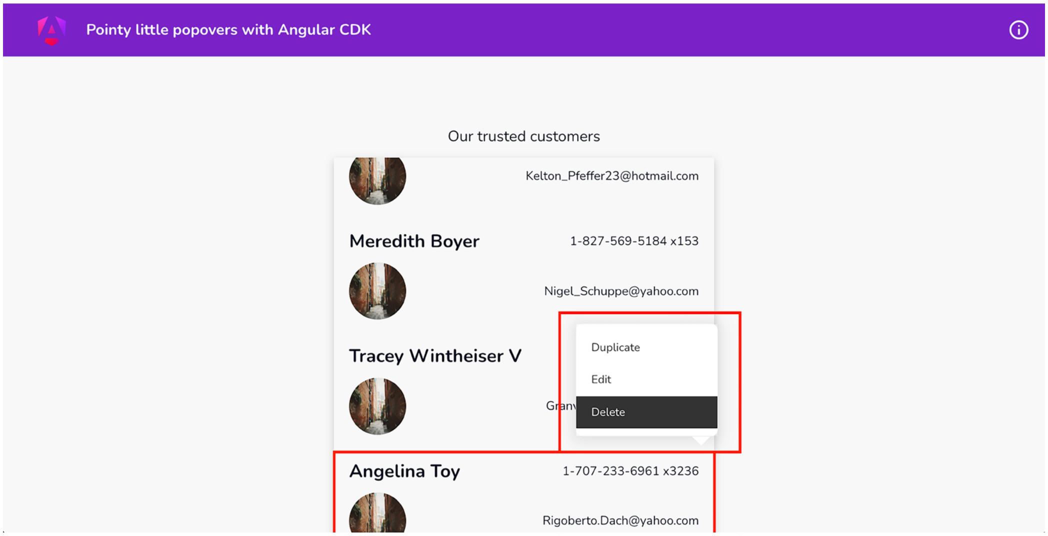

Pointy little popovers with Angular CDK
Our trusted customers
Kelton_Pfeffer23@hotmail.com
Meredith Boyer
1-827-569-5184 x153
Nigel_Schuppe@yahoo.com
Tracey Wintheiser V
Duplicate
Edit
Delete
Angelina Toy
1-707-233-6961 x3236
Rigoberto.Dach@yahoo.com

Figure 9.10: Drop-down arrow pointing at the correct list item (pointing downward)

Great! You now know how to use the Angular CDK to work with overlays to create custom pop-up/drop-down menus. Moreover, you now know how to quickly implement the pointy arrows on the menu using a custom directive. See the next section to understand how it all works.

How it works...

Implementing an overlay using the Angular CDK Overlay API includes a couple of pieces to work with. We first have to import the `OverlayModule` class in our `UserListComponent` class's imports. We do this because the `OverlayModule` contains the directives we are using in the application: for example, the `CdkConnectedOverlay` directive and the `CdkOverlayOrigin` directive. After importing the module for creating an overlay, we need to have both an overlay (the menu in our case) and an overlay origin (which opens/shows the menu). In this recipe, since we're using the overlay to create a popover menu for each list item, we use the `cdkOverlayOrigin` directive on the `<app-users-list-item>` elements. Notice that the `<app-users-list-item>` elements are being rendered through the `*ngFor` directive.

Therefore, in order to know which item was clicked or precisely which item we need to show the popover for, we create a `#menuTrigger` template variable on each list item element, and you'll notice that we also bind the `(click)` event on the list items to call the `openMenu` method, passing this `menuTrigger` template variable into it.

Now, if you have noticed the `openMenu` method in the `users-list.component.ts` file, it looks like this:

```
openMenu($event: Event, itemTrigger: CdkOverlayOrigin) {
    if ($event) {
      $event.stopImmediatePropagation();
    }
    this.popoverMenuTrigger = itemTrigger;
    this.menuShown = true;
}
```

Notice that we assign the `itemTrigger` property to our class's `popoverMenuOrigin` property. This is because this `popoverMenuOrigin` property is being bound with the actual overlay in our template. You can also see that we set the `menuShown` property to `true`, and this is because it will decide whether the overlay should be shown or hidden.

Now, let's see the code for the actual overlay, as follows:

```
<ng-template cdkConnectedOverlay
[cdkConnectedOverlayOrigin]="popoverMenuOrigin!"
  [cdkConnectedOverlayOpen]="menuShown" [cdkConnectedOverlayHasBackdrop]="true"
  (backdropClick)="closeMenu()"
  [cdkConnectedOverlayPositions]="menuPositions"
  appPopoverPositionalClass targetSelector=".menu-popover" inverseClass="menu-
popover--up"
  [originY]="popoverMenuPosition.originY"
    (positionChange)="popoverPositionChanged($event)"
  cdkConnectedOverlayPanelClass="menu-popover"
  >
  ...
</ng-template>
```

Let's discuss each of the `cdkConnectedOverlay` directive's attributes, one by one:

- The `cdkConnectedOverlay` attribute: This is the actual overlay directive that makes the `<ng-template>` element an Angular CDK overlay.
- The `[cdkConnectedOverlayOrigin]` attribute: This tells the Overlay API what the origin of this overlay is. This is to help the CDK decide where to position the overlay when opened.
- The `[cdkConnectedOverlayOpen]` attribute: This decides whether the overlay should be shown or not.

- The `[cdkConnectedOverlayHasBackdrop]` attribute: This decides whether the overlay should have a backdrop or not—that is, if it has a backdrop, the user shouldn't be able to click anything else apart from the overlay when it is open.
- The `(backdropClick)` attribute: This is the event handler for when we click the backdrop. In this case, we're setting the `menuShown` property to `false`, which hides/closes the overlay.
- The `[cdkConnectedOverlayPositions]` attribute: This provides the positioning configuration to the Overlay API. It is an array of preferred positions that defines whether the overlay should be shown below the origin, on top of the origin, on the left, on the right, how far from the origin, and so on using properties such as `originX`, `originY`, `overlayX`, and `overlayY`.
- The `[cdkConnectedOverlayPanelClass]` attribute: A **CSS** class to be applied to the generated overlay. This is used for styling.

With all of the attributes set correctly, we are able to see the overlay working when tapping the list items. *"But what about the pointy arrows, Ahsan?"* Well, hold on! We'll discuss them too.

So, the Angular CDK Overlay API already has a lot of things covered, including where to position the overlay based on the available space, and since we want to show the pointy arrows, we'll have to analyze whether the overlay is being shown above the item or below the item. By default, we have the following styles set in the `src/styles.scss` file to show the pointy arrow below the popover:

```
.menu-popover {
  ...
  &::before {
    top: -10px;
    border-width: 0px 10px 10px 10px;
    border-color: transparent transparent white  transparent;
    position: absolute;
    content: '';
    right: 5%;
    border-style: solid;
  }
  &--up {...}
  &__list {...}
}
```

And then, we have the `--up` modifier class, as follows, to show the overlay *above* the popover:

```
.menu-popover {
  ...
  &::before {...}
  &--up {
    transform: translateY(-20px);
```

```
      &::before {
        top: unset !important;
        transform: rotate(180deg);
        bottom: -10px;
      }
    }
    &__list {...}
  }
```

Notice in the preceding code snippet that we rotate the arrow to **180deg** to invert its pointer.

Now, let's talk about how and when this `--up` modifier class is applied. We have created a custom directive named `appPopoverPositionalClass`. This directive is also applied to the `<ng-template>` element we have for the overlay—that is, this directive is applied with the `cdkConnectedOverlay` directive and expects the following input attributes:

- The `appPopoverPositionalClass` attribute: The actual directive selector.
- The `targetSelector` attribute: The query selector for the element that is generated by the Angular CDK Overlay API. Ideally, this should be the same as what we use in `cdkConnectedOverlayPanelClass`.
- The `inverseClass` attribute: The class to be applied when the vertical position (`originY`) of the overlay is changed—that is, from `"top"` to `"bottom"`, and vice versa.
- The `originY` attribute: The `originY` position of the overlay at the moment. The value is either `"top"` or `"bottom"`, based on the overlay position.

We have a `(positionChange)` listener on the CDK Overlay `<ng-template>` element that triggers the `popoverPositionChanged` method as soon as the overlay position changes. Notice that inside the `popoverPositionChanged` method, upon getting a new position, we update the `popover.originY` property that is updating `menuPopoverOrigin.originY`, and then we're also passing `menuPopoverOrigin.originY` as the `[originY]` attribute to our `appPopoverPositionalClass` directive. Since we're passing it to the directive, the directive knows if the overlay position is `"top"` or `"bottom"` at any particular time. How? Because we're using the `ngOnChanges` life cycle hook in the directive to listen to the `originY` attribute/input, and as soon as we get a different value for `originY`, we either add the value of `inverseClass` as a CSS class to the `Overlay` element or remove it based on the value of the `originY` attribute. Also, based on the applied CSS classes, the direction of the popover arrow is decided for the overlay.

See also

- Angular CDK Overlay API: `https://material.angular.io/cdk/overlay/overview`
- `CdkOverlayOrigin` directive documentation: `https://material.angular.io/cdk/overlay/api#CdkOverlayOrigin`

Input coercion utilities from the Angular CDK

This is one of those cool things that you see and think, “Oh, that could’ve been handy if I knew about this before.” I absolutely thought that about these coercion utilities from the Angular CDK. There have been many times when you have a number value as an input in an Angular component and you are forced to use the square bracket notation (`[myNumberInput]="numberValue"`) because if you use `"myNumberInput="numberValue"`, it is interpreted as a literal string `"numberValue"` instead of a variable. The coercion utilities from the Angular CDK are exactly what we need here.

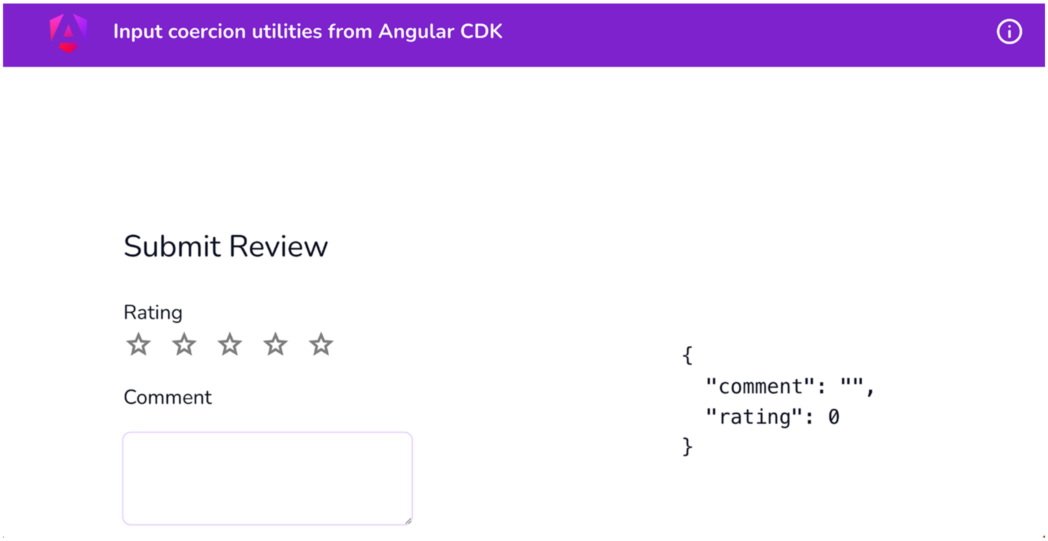
Getting ready

The app that we are going to work with resides in `start/apps/chapter09/ng-cdk-coercion` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve ng-cdk-coercion
   ```

   This should open the app in a new browser tab, and you should see the following:



```
{
  "comment": "",
  "rating": 0
}
```

Figure 9.11: using-cdk-coercion running on http://localhost:4200

Now that we have the app running locally, let’s see the steps of the recipe in the next section.

How to do it…

We have a rating component that takes a few inputs, including “value” and “disabled.” However, we can’t really use them like usual attributes and have to use square brackets notation for the Angular attribute bindings for both of them. We’ll fix that in the recipe. Let’s begin:

1. First of all, let's say we want to add a new feature to the ratings component—that is, having a max number for the rating. Right now, it defaults to `5`. So we'll create a new input in the `rating.component.ts` file as follows:

```
...
export class RatingComponent {
  @Input() value = 2;
  @Input() disabled = false;
  @Input() max = 5;
  ...
  get renderArr() {
    return new Array(this.max).fill(0).map((_, index) =>
      index + 1);
  }
  ...
}
```

2. Now, we'll provide the max input to the rating component from the home component. We'll set its value to `8`. But we'll use the usual HTML notation to supply the value. Update the usage of the `<app-rating-component>` in the `home.component.html` file, as follows:

```
        <app-rating
           [value]="reviewForm.controls.rating.value" max="8"
           [disabled]="ratingDisabled"
           (valueChanged)="applyRating($event)">
         </app-rating>
```

You'll notice immediately that the IDE throws an error as we can't assign a string value to a number input.

3. We'll now use the coercion utilities from the Angular CDK to solve this issue. Update the `rating.component.ts` file to use the `NumberInput` type and the `coerceNumberProperty` method as follows:

```
...
import { NumberInput, coerceNumberProperty } from '@angular/cdk/
coercion';
...
export class RatingComponent {
  @Input() value = 2;
  @Input() disabled = false;
  @Input() get max () {
    return this._max;
  }
  set max (val: NumberInput) {
    this._max = coerceNumberProperty(val);
  }
  private _max = 5;
  @Output() valueChanged: EventEmitter<number> = new
    EventEmitter();
  ...
}
```

If you check the `home.component.html` file again, you'll see that the error is gone and the app compiles without any issues.

4. Let's make the `disabled` property coerced as well. We'll try to use the double-curly-braces `{{}}` notation for this input. Update the usage of the `<app-rating>` element in the `home.component.html` file as follows:

```
        <app-rating
          [value]="reviewForm.controls.rating.value"
          max="8"
          disabled="{{ratingDisabled}}"
          (valueChanged)="applyRating($event)">
        </app-rating>
```

You'll see an error in both the IDE and the compilation: `Type 'string' is not assignable to type 'boolean'.`

5. To fix the issue, let's use the `BooleanInput` and `coerceBooleanProperty` method in the `rating.component.ts` file as follows:

```
...
import { BooleanInput, NumberInput, coerceBooleanProperty,
coerceNumberProperty } from '@angular/cdk/coercion';
...
export class RatingComponent {
  @Input() value = 2;
  @Input() get disabled () {
    return this._disabled;
  }
  set disabled (val: BooleanInput) {
    this._disabled = coerceBooleanProperty(val);
  }
  private _disabled = false;
  @Input() get max () {...}
  ...
}
```

And voilà! Your code should be compiled without errors, and you should be able to provide values without the forced square-bracket attribute binding notation. What's great is that even if you just add the `disabled` attribute to the `<app-rating>` element, it will disable the component, which is what happens when we apply the `disabled` attribute to a regular `input` element or `textarea`.

How it works...

In the recipe, we used two different methods from the Angular CDK's coercion utilities. The `coerceNumberProperty` method and the `coerceBooleanProperty` method. Let's first have a look at the `coerceNumberProperty` method along with the `NumberInput` type.

The `NumberInput` type resolves to being `string | number | null | undefined`, which makes it possible for us to provide the values as any of those types. And when we don't use the square-bracket notation for attribute binding, the value is considered as a **string**. This seems to be covered by the `NumberInput` type. The `coerceNumberProperty` method takes any value and converts it into a number. The function also takes a second optional argument as the fallback value.

The usages of the `BooleanInput` type and the `coerceBooleanProperty` method are very similar to the `NumberInput` type and `coerceNumberProperty` respectively. The difference, however, is that the `coerceBooleanProperty` method takes any value and converts it into a boolean. The logic behind this method is a bit different as it looks for the following condition:

```
value != null && `${value}` !== 'false'
```

Which means if the attribute is provided without a value, it is going to be an empty string. And even though empty string is a falsy value, with the coerceBooleanProperty, an attribute without a value (empty string) will resolve as a truthy value.

See also

- CDK coercion documentation: `https://material.angular.io/cdk/coercion/overview`

Using the CDK Drag and Drop API to move items from one list to another

Have you ever used the Trello board app, or maybe other apps that also allow you to drag and drop list items from one list to another? Well, you can do this easily using the Angular CDK, and in this recipe, you'll learn about using the **Angular CDK Drag and Drop API** to move items from one list to another. You'll also learn how to reorder lists.

Getting ready

The app that we are going to work with resides in `start/apps/chapter09/ng-cdk-drag-drop` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-cdk-drag-drop to serve the project.
```

This should open the app in a new browser tab, and you should see the following:

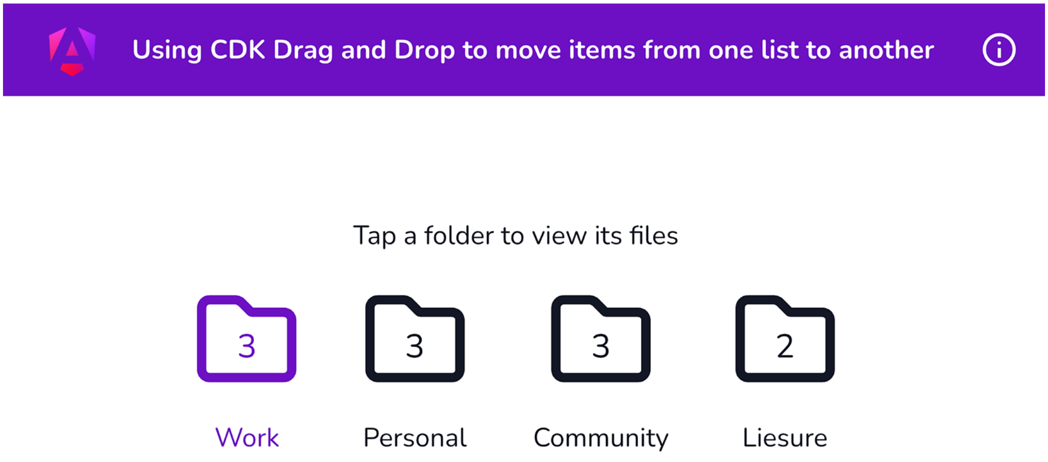

Using CDK Drag and Drop to move items from one list to another
Tap a folder to view its files
3
Work
3
Personal
3
Community
2
Liesure

Figure 9.12: The ng-cdk-drag-drop app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

For this recipe, we have an interesting app that has some folders and files. We're going to implement the drag-and-drop functionality for the files to be dragged to other folders, which should update the folder's file count instantly, and we should be able to see the file in the new folder as well. Let's get started.

1. First of all, we need to import `DragDropModule` because we use the `CdkDrag` and the `CdkDropList` directives in our app which belong to it. We will import the module into the `imports` array of our `FoldersListComponent` class. Modify the `folders-list.component.ts` file, as follows:

```
...
import { DragDropModule } from '@angular/cdk/drag-drop';
@Component({
  ...
  imports: [CommonModule, FileComponent, FolderComponent,
DragDropModule],
  standalone: true
})
...
```

2. Now, we'll apply the `cdkDrag` directive to each of our files and will apply the `cdkDropList` directive to each of the folders. Update the `folders-list.component.html` file, as follows:

```
<div class="folders">
  ...
  <div class="folders__list">
```

```
      <app-folder
        cdkDropList
        ...
        [folder]="folder"
      >
      </app-folder>
    </div>
    <div class="folders__selected-folder-files"
      *ngIf="selectedFolder">
      <div>
        <app-file
          cdkDrag
          *ngFor="let file of selectedFolder.files"
          [file]="file"
        ></app-file>
      </div>
    </div>
  </div>
```

3. We'll also enable reordering of the files within a folder by adding the `cdkDropList` directive to the container elements for the files, as follows:

```
<div class="folders">
  ...
  <div class="folders__selected-folder-files"
    *ngIf="selectedFolder">
    <div cdkDropList>
      <app-file ...></app-file>
    </div>
  </div>
</div>
```

4. We'll now define the origin of the drag-and-drop interaction by specifying the `[cdkDragData]` attribute on each `<app-file>` element and the `[cdkDropListData]` attribute on each `<app-folder>` element, and on the files container as well. Update the template again, as follows:

```
<div class="folders">
  ...
  <div class="folders__list">
    <app-folder
      cdkDropList
      [cdkDropListData]="folder.files"
      ...
      >
      </app-folder>
    </div>
    <div class="folders__selected-folder-files"
      *ngIf="selectedFolder">
      <div
        cdkDropList
        [cdkDropListData]="selectedFolder.files"
      >
        <app-file
          cdkDrag
          [cdkDragData]="file"
          ...
        ></app-file>
      </div>
    </div>
  </div>
```

5. We now need to implement what happens when the file is dropped. To do so, we'll use the `(cdkDropListDropped)` event handler. Update the template, as follows:

```
  <div class="folders">
    ...
    <div class="folders__list">
      <app-folder
        cdkDropList
        [cdkDropListData]="folder.files"
        (cdkDropListDropped)="onFileDrop($event)"
        ...
      >
      </app-folder>
    </div>
    <div class="folders__selected-folder-files"
      *ngIf="selectedFolder">
      <div
        cdkDropList
        [cdkDropListData]="selectedFolder.files"
        (cdkDropListDropped)="onFileDrop($event)"
      >
        ...
      </div>
    </div>
  </div>
```

6. Finally, we need to implement the `onFileDrop` method. Update the `folders-list.component.ts` file, as follows:

```
...
import { IFile, IFolder } from '../interfaces';
...
import { CdkDragDrop, DragDropModule, moveItemInArray, transferArrayItem
} from '@angular/cdk/drag-drop';
...
export class FoldersListComponent {
  ...
  toggleFolderSelect(folder: IFolder) {...}

  onFileDrop(event: CdkDragDrop<IFile[]>) {
    if (event.previousContainer === event.container) {
      moveItemInArray(
        event.container.data, event.previousIndex,
        event.currentIndex
      );
    } else {
      transferArrayItem(
        event.previousContainer.data, event.container.data,
        event.previousIndex, event.currentIndex
      );
    }
  }
}
```

   The `moveItemInArray` method and the `transferArrayItem` method are built-in from the Angular CDK. This makes our implementation a breeze.

7. In order to implement drag-and-drop previews, we need to enclose the droplists and the draggable items into an element with the `cdkDropListGroup` directive. Update the `folders-list.component.html` file and apply the directive to the element with the `folders` class (i.e., at the top level), as follows:

```
<div class="folders" cdkDropListGroup>
...
</div>
```

8. To apply a custom drag preview, we use a custom element with the `*cdkDragPreview` directive applied to it. Update the `folders-list.component.html` file, as follows:

```
<div class="folders" cdkDropListGroup>
  ...
  <div class="folders__selected-folder-files"   *ngIf="selectedFolder">
    <div
      cdkDropList
      ...
    >
      <app-file
        cdkDrag
        ...
      >
        <fa-icon
          class="file-drag-preview"
          *cdkDragPreview
          [icon]="file.icon!"
        ></fa-icon>
      </app-file>
    </div>
  </div>
</div>
```

9. For the `fa-icon` component to work, we also need to import the `FontAwesomeModule` in the `FoldersListComponent` class. Update the file below as follows:

```
...
import { FontAwesomeModule } from '@fortawesome/angular-fontawesome';
@Component({
  ...
  imports: [CommonModule, FileComponent, FolderComponent, DragDropModule,
FontAwesomeModule],
  standalone: true
})
...
```

10. We'll also need some styles for the drag-and-drop previews. Update the `folders-list.component.scss` file, as follows:

```
$folder-bg: #f5f5f5;
$file-preview-transition: transform 250ms cubic-bezier(0,
  0, 0.2, 1);
.folders {...}
```

```
.file-drag-preview {
  padding: 10px 20px;
  background: transparent;
  font-size: 32px;
}
.file-drop-placeholder {
  min-height: 60px;
  transition: $file-preview-transition;
  display: flex;
  align-items: center;
  justify-content: center;
  font-size: 32px;
}
```

11. Let's also add some styles to make sure the other list items move smoothly when reordering the items within a folder. Since we need to style the elements from the Angular CDK, and they have styles encapsulation, we need to add the styles in our global `scss` file. Update the `folders-list.component.scss` file further, as follows:

```
...
* {
  user-select: none;
}

.cdk-drop-list-dragging .cdk-drag {
  transition: transform 250ms cubic-bezier(0, 0, 0.2, 1);
}

.cdk-drag-animating {
  transition: transform 300ms cubic-bezier(0, 0, 0.2, 1);
}
```

12. Now, we need to create a drop preview (a placeholder that appears until we release the dragged file) template as well. For this, we use the `*cdkDragPlaceholder` directive on the `preview` element. Update the `folders-list.component.html` file, as follows:

```
<div class="folders" cdkDropListGroup>
  ...
  <div class="folders__selected-folder-files" *ngIf="selectedFolder">
    <div cdkDropList ...>
      <app-file cdkDrag ...>
        <fa-icon class="file-drag-preview"
          *cdkDragPreview ... ></fa-icon>
        <div class="file-drop-placeholder"
          *cdkDragPlaceholder>
        <fa-icon [icon]="upArrow"></fa-icon>
      </div>
      </app-file>
    </div>
  </div>
</div>
```

13. Finally, let's create an `upArrow` property using the `faArrowAltCircleUp` icon from the `@fortawesome` package. Update the `folders-list.component.ts` file, as follows:

```
...
import { faArrowAltCircleUp } from '@fortawesome/free-regular-svg-icons';
...
export class FoldersListComponent {
  folders = APP_DATA;
  upArrow = faArrowAltCircleUp;
  selectedFolder: IFolder | null = null;
  ...
}
```

And boom! We now have a seamless **User Experience** (**UX**) for the entire drag-and-drop flow. Like it? Make sure that you share a snapshot on your X (Twitter) and tag me at **@codewith_ahsan**.

Now that we've finished the recipe, let's see in the next section how it all works.

How it works...

There were a couple of interesting directives in this recipe, and we'll go through them all one by one. First of all, as good Angular developers, we import the `DragDropModule` class into the `imports` array of our `standalone component` named `FoldersListComponent`, just to make sure we don't end up with errors. Then, we start making the files draggable. We do this by adding the `cdkDrag` directive to each file element by applying the `*ngFor` directive to it. This tells the Angular CDK that this element will be dragged and, therefore, the Angular CDK binds different handlers to each element to be dragged.

> IMPORTANT NOTE
>
> Angular components by default are not block elements. Therefore, when applying the `cdkDrag` directive to an Angular component such as the `<app-file>` component, it might restrict the animations from the CDK being applied when we're dragging the elements. In order to fix this, we need to set a `display: block;` for our component elements. Notice that we're applying the required styles in the `folders-list.component.scss` file (line 25) for the `.folders__selected-folder-files__file` class.

After configuring the drag elements, we use the `cdkDropList` directive for each container DOM element where we're supposed to drop the file. In our recipe, that is each folder that we see on the screen, and we can also reorder the files within a folder. Therefore, we apply the `cdkDropList` directive to the wrapper element of the currently displayed files, as well as to each `<app-folder>` item with the `*ngFor` looping over the `folders` array.

Then, we specify the **data** that we're dragging by specifying `[cdkDragData]="file"` for each draggable file. This helps us identify it in the later process when we drop it either within the current folder or within other folders. We also specify in which array this dragged item will be added when dropped upon the particular list, and we do this by specifying `[cdkDropListData]="ARRAY"` statements for the elements that we've applied the `cdkDropList` directive to. When the Angular CDK combines the information from the `cdkDragData` and the `cdkDropListData` attributes, it can easily identify if the item was dragged and then dropped within the same list or in another list.

To handle what happens when we drop the dragged file, we use the `(cdkDropListDropped)` method from the Angular CDK on the element with the `cdkDropList` directive. We take the `$event` emitted from the CDK and pass it to our `onFileDrop` method. What's great is that within the `onFileDrop` method, we use the `moveItemInArray` and `transferArrayItem` helper methods from the Angular CDK, with a really simple bit of logic to compare the containers. That is, the Angular CDK provides us with enough information that we can get away with the whole functionality really easily.

Toward the end of the recipe, we customize how our drag preview should look when we are dragging a file using a custom template, by using the `*cdkDragPreview` directive on it. This tells the Angular CDK to not render it right away but to show it with the mouse when we start dragging a file. For our recipe, we only show the icon of the file as the drag preview. And finally, we also customize the drop preview (or drag placeholder) usingthe `*cdkDragPlaceholder` directive, which shows a transparent rectangle with an upward-arrow icon to reflect where the item is going to be added when dropped. Of course, we had to add some custom styles for both the drag preview and the drop preview.

See also

- Angular CDK Drag and Drop documentation: `https://material.angular.io/cdk/drag-drop/overview`

Creating a multi-step game with the CDK Stepper API

If you try finding examples of the CDK Stepper API over the internet, you'll find a bunch of articles revolving around creating multi-step forms using the CDK Stepper API, but since it is a stepper at its base, it can be used for various use cases. In this recipe, we're going to build a guessing game using the Angular CDK Stepper API, in which the user will guess what the output of a rolled dice will be.

Getting ready

The app that we are going to work with resides in `start/apps/chapter09/ng-cdk-stepper` inside the cloned repository:

1. Open the code repository in your code editor.

2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-cdk-stepper
```

This should open the app in a new browser tab, and you should see the following:

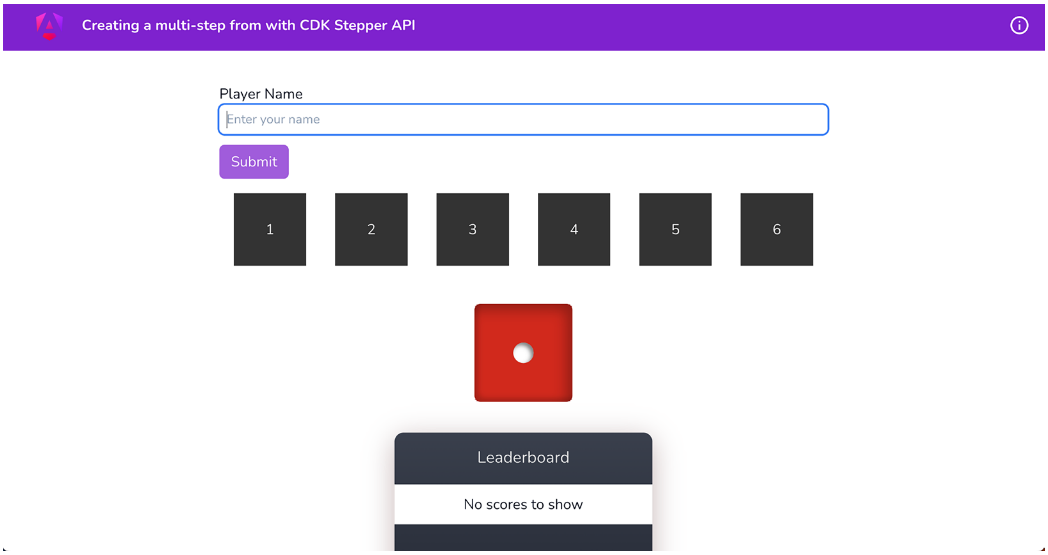

Creating a multi-step from with CDK Stepper API
Player Name
Enter your name
Submit
1
2
3
4
5
6
Leaderboard
No scores to show

Figure 9.13: The ng-cdk-stepper app running on http://localhost:4200

Now, let's look at how to create a multi-step game with the CDK Stepper API in the next section.

How to do it...

We have a really simple yet interesting application at hand that has a couple of components built already, including the `dice component`, the `value-guess component`, and the `leaderboard component`. We'll create this game as a multi-step game using the **Stepper API**. Proceed as follows:

1. First, import the `CdkStepperModule` class from the `@angular/cdk` package in the `game.component.ts` file, as follows:

```
...
import { CdkStepperModule } from '@angular/cdk/stepper';
@Component({
  ...
  imports: [
    ...
    DiceComponent,
    CdkStepperModule
  ],
})
...
```

2. Let's create our stepper component now. Run the following command in the workspace root folder:

```
cd start && nx g c game-stepper --directory apps/chapter09/ng-cdk-
stepper/src/app/components/game-stepper
```

   Choose the `@nx/angular:component` schematic when asked and select the "As provided" option.

3. To make our component a `CdkStepper`, we need to provide it using the `CdkStepper` token and have to extend our component class from `CdkStepper` as well. We can remove the `constructor`, the `OnInit` implementation, and the `ngOnInit` method. Modify the `game-stepper.component.ts` file, as follows:

```
...
import { CdkStepper, CdkStepperModule } from '@angular/cdk/stepper';
@Component({
  ...
  imports: [CommonModule, CdkStepperModule],
  ...
  providers: [{ provide: CdkStepper, useExisting:
    GameStepperComponent }],
})
export class GameStepperComponent extends CdkStepper {
}
```

   The `CDKStepper` provides us helpful methods like `next`, `previous`, and `reset` to navigate between the steps, and the `selectedIndexChange` event emitter to identify when a step is changed.

4. Let's add the template for our `<game-stepper>` component. We'll start by adding the header that will show the step label. Update your `game-stepper.component.html` file, as follows:

```
<div class="game-stepper">
  <header>
    <h3 *ngIf="selected">
      <ng-container
        *ngIf="selected.stepLabel; else showLabelText"
        [ngTemplateOutlet]="selected.stepLabel.template"
      >
      </ng-container>
      <ng-template #showLabelText>
        {{ selected.label }}
      </ng-template>
    </h3>
  </header>
</div>
```

5. Now, we'll add the template to show our main content for the selected step—this is pretty simple to do. We need to add a `div` with the `[ngTemplateOutlet]` attribute, where we'll show the content. Update the `game-stepper.component.html` file, as follows:

```
<div class="game-stepper">
  <header>...</header>
  <section class="game-stepper__content">
  <div [ngTemplateOutlet]="selected ? selected.content :
    null"></div>
</section>
  ...
</div>
```

6. Finally, we'll add a footer element that'll contain the navigation buttons for our stepper— that is, we should be able to jump to the next and the previous step using those navigation buttons. Update the `game-stepper.component.html` file further, as follows:

```
<div class="game-stepper">
  <header>...</header>
  <section class="game-stepper__content">...</section>
  <footer class="game-stepper__navigation">
    <button *ngIf="steps.get(selectedIndex - 1)"
    class="game-stepper__navigation__button btn btn
    -primary" cdkStepperPrevious>
      &larr;
    </button>
    <span class="flex-1"></span>
    <button *ngIf="steps.get(selectedIndex + 1)"
      class="game-stepper__navigation__button btn btn-
      primary" cdkStepperNext>
      &rarr;
    </button>
  </footer>
</div>
```

7. Let's add some styles to our `game-stepper` component. Modify the `game-stepper.component.scss` file, as follows:

```
.game-stepper {
  display: flex;
  flex-direction: column;
  align-items: center;
  &__navigation {
    width: 100%;
    display: flex;
    align-items: center;
    justify-content: space-between;
    > button {
      margin: 0 8px;
    }
  }

  &__content {
    min-height: 150px;
    display: flex;
    justify-content: center;
    align-items: center;
    flex-direction: column;
  }
  header,
  footer {
    margin: 10px auto;
  }
}
```

8. Now we want to use the `game stepper` component inside the `game` component. To do this, first import it in the `GameComponent` class's `imports` array as follows:

```
...
import { GameStepperComponent } from '../components/game-stepper/game-stepper.component';
@Component({
  ...
  imports: [
    ...
    GameStepperComponent
  ],
})
```

9. We'll now wrap our entire template in the `game.component.html` file with the `<app-game-stepper>` component. Update the file, as follows:

```
<app-game-stepper>
  <form (ngSubmit)="submitName()"
    [formGroup]="nameForm">...</form>
  <app-value-guesser></app-value-guesser>
  <app-dice></app-dice>
```

```
    <app-leader-board></app-leader-board>
  </app-game-stepper>
```

10. We'll now modify our `game.component.html` file to break down the inner template into steps. For that, we'll use the `<cdk-step>` element to wrap around the content for each step. Update the file, as follows:

```
  <app-game-stepper>
    <cdk-step>
      <form (ngSubmit)="submitName()"
        [formGroup]="nameForm">
        ...
      </form>
    </cdk-step>
    <cdk-step>
      <app-value-guesser></app-value-guesser>
      <app-dice></app-dice>
    </cdk-step>
    <cdk-step>
      <app-leader-board></app-leader-board>
    </cdk-step>
  </app-game-stepper>
```

11. Now, we'll add a label for each step. We need to add an `<ng-template>` element inside each `<cdk-step>` element. Update the `game.component.html` file, as follows:

```
  <app-game-stepper>
    <cdk-step>
      <ng-template cdkStepLabel>Enter your name</ng-
        template>
      <form (ngSubmit)="submitName()"     [formGroup]="nameForm">
        ...
      </form>
    </cdk-step>
    <cdk-step>
      <ng-template cdkStepLabel>
        Guess what the value will be when the die is rolled
      </ng-template>
      <app-value-guesser></app-value-guesser>
      <app-dice></app-dice>
    </cdk-step>
    <cdk-step>
      <ng-template cdkStepLabel>Results</ng-template>
    <app-leader-board></app-leader-board>
  </cdk-step>
</app-game-stepper>
```

If you refresh the app, you should see the first step as the visible step, as well as the bottom navigation button, as follows:

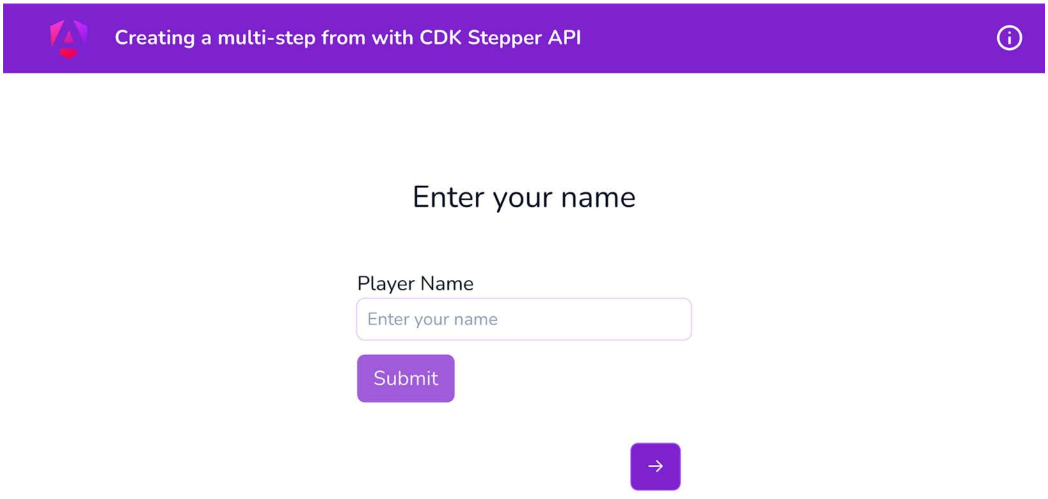

Creating a multi-step from with CDK Stepper API
Enter your name
Player Name
Enter your name
Submit

Figure 9.14: The first step and the navigation button using CDKStepper

12. Now, we need to make sure that we can only move forward to the second step once we have entered a name in the first step. Make the following changes to the `game.component.html` file:

```
<app-game-stepper [linear]="true">
  <cdk-step [completed]="!!nameForm.get('name')!.value">
    <ng-template cdkStepLabel> Enter your
      name</ng-template>
    <form (ngSubmit)="submitName()"
      [formGroup]="nameForm">
      <div class="mb-3" *ngIf="nameForm.get('name') as
        nameControl">
        ...
      </div>
      <button ← REMOVE THIS
        type="submit"
        [disabled]="!nameForm.valid"
        class="btn btn-primary"
      >
        Submit
      </button>
    </form>
  </cdk-step>
  ...
</app-game-stepper>
```

13. We also need to disable the next button on the first step until we have entered a value for the player's name. To do so, update the `game-stepper.component.html` file—specifically, the element with the `cdkStepperNext` attribute—as follows:

```
<section class="game-stepper">
  ...
  <footer class="game-stepper__navigation">
    ...
    <button
      class="game-stepper__navigation__button btn
        btn-primary"
      cdkStepperNext
      [disabled]="!selected!.completed"
      [style.visibility]="steps.get(selectedIndex + 1) ?
        'visible' : 'hidden'"
    >
      &rarr;
    </button>
  </footer>
</section>
```

14. To handle the case when the user provides the name and hits the *Enter* key, resulting in the form submission, we can move to the next step using a `@ViewChild()` in the `GameComponent` class. Modify the `game.component.ts` file as follows, and try entering the name and then pressing the *Enter* key:

```
import { CommonModule } from '@angular/common';
import { Component, ViewChild } from '@angular/core';
import { CdkStepper, CdkStepperModule } from '@angular/cdk/stepper';
...
export class GameComponent {
  @ViewChild(CdkStepper) stepper!: CdkStepper;
  ...
  submitName() {
    this.stepper.next();
  }
}
```

15. Now, let's write the flow for guessing the number. Update the `game.component.ts` file, as follows:

```
...
import { IDiceSide } from '../interfaces/dice.interface';
@Component({...})
export class GameComponent implements OnInit {
  @ViewChild(CdkStepper) stepper: CdkStepper;
  @ViewChild(DiceComponent) diceComponent!: DiceComponent;
  @ViewChild(ValueGuesserComponent)
  valueGuesserComponent!: ValueGuesserComponent;
  guessedValue: number | null = null;
  isCorrectGuess: null | boolean = null;
  ...
  submitName() {...}
  rollTheDice(guessedValue: number) {
    this.isCorrectGuess = null;
    this.guessedValue = guessedValue;
    this.diceComponent.rollDice();
  }
  showResult(diceSide: IDiceSide) {
    this.isCorrectGuess = this.guessedValue ===
      diceSide.value;
  }
}
```

16. Now that we have the functions in place, let's update the template to listen to the event listeners from the `<app-value-guesser>` and `<app-dice>` components and act accordingly. We'll also add elements with class alerts to show messages in case of a successful or wrong guess. Update the `game.component.html` file, as follows:

```
<app-game-stepper [linear]="true">
  <cdk-step [completed]="!!nameForm.get('name').value">
    ...
  </cdk-step>
  <cdk-step [completed]="isCorrectGuess !== null">
    <ng-template cdkStepLabel
      >Guess what the value will be when the die is
       rolled</ng-template
    >
    <app-value-guesser [rolling]="rolling" (valueGuessed)=
      "rollTheDice($event)"></app-value-guesser>
    <app-dice (diceRolling)="rolling = $event;"
      (diceRolled)="showResult($event)"></app-dice>
      <ng-container [ngSwitch]="isCorrectGuess">
        <div class="alert alert-success"
          *ngSwitchCase="true">
          You rock {{ nameForm.get('name')!.value }}! You got
          50 points
        </div>
        <div class="alert alert-danger"
          *ngSwitchCase="false">
          Oops! Try again!
        </div>
      </ng-container>
    </cdk-step>
    <cdk-step>...</cdk-step>
  </app-game-stepper>
```

17. Finally, we need to populate the leaderboards. Update the `game.component.ts` file to use the `LeaderboardService` class, as follows:

```
...
import { Component, ViewChild, inject } from '@angular/core';
...
import { LeaderboardService } from '../services/leaderboard.service';
import { IScore } from '../interfaces/score.interface';
...
export class GameComponent {
  ...
  leaderboardService = inject(LeaderboardService);
  scores: IScore[] = this.leaderboardService.getScores();
  ...
  showResult(diceSide: IDiceSide) {
    this.isCorrectGuess = this.guessedValue ===
      diceSide.value;
    const userName = this.nameForm.controls.name.value as
      string;
    if (!this.isCorrectGuess) {
      return;
    }
    this.scores = this.leaderboardService.setScores({
      name: userName,
      score: 50,
    });
  }
}
```

18. Now, update the `game.component.html` file to pass the scores as an attribute to the `<app-leader-board>` component, as follows:

```
<app-game-stepper [linear]="true">
  <cdk-step [completed]="!!nameForm.get('name')!
    .value"></cdk-step>
  <cdk-step [completed]="isCorrectGuess !== null">
  </cdk-step>
  <cdk-step>
    <ng-template cdkStepLabel>Results</ng-template>
    <app-leader-board [scores]="scores"></app-leader-
    board>
  </cdk-step>
</app-game-stepper>
```

If you refresh the app now and play the game, you should be able to see the leaderboard, as follows:

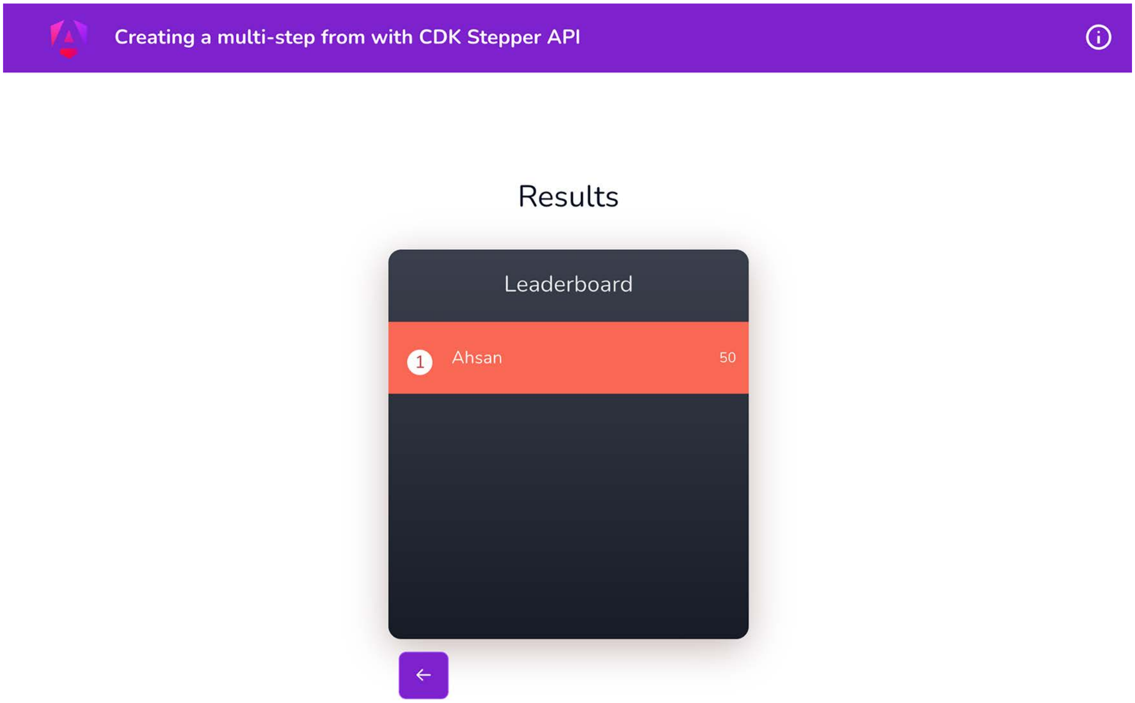

Creating a multi-step from with CDK Stepper API
Results
Leaderboard
1
Ahsan
50
←

Figure 9.15: Displaying results in the leaderboard at Step 3

Phew! That was a *long* recipe! Well, perfection requires time and dedication. Feel free to use this game yourself or even with your friends, and if you improve it, do let me know on my socials.

Now that you've finished the recipe, see the next section on how this works.

How it works...

There are a lot of moving parts in this recipe, but they're super easy. First, we import the `CdkStepperModule` class into the `imports` array of our `GameComponent` class. Then, we create a component that extends the `CdkStepper` class. The reason for extending the `CdkStepper` class is to be able to create this `GameStepperComponent` component so that we can create a reusable template with some styles, and even some custom functionality.

To start using the `GameStepperComponent` component, we wrap the entire template within the `<app-game-stepper>` element in the `game.component.html` file. Since the component extends the `CdkStepper` API, we can use all the functionality of the `CdkStepper` component here. For each step, we use the `<cdk-step>` element from the CDK and wrap the template of the step inside it. Notice that in the `game-stepper.component.html` file, we use the `[ngTemplateOutlet]` attribute for both the step's label and the step's actual content. This is a reflection of how amazing the CDK Stepper API is. Since our `GameStepperComponent` class extends the `CDKStepper` class, it automatically generates a `label` property and the `content` property for each step based on the values/template we provide for each `<cdk-step>` element in the `game.component.html` file. Since we provide an `<ng-template cdkStepLabel>` inside each `<cdk-step>` element in the `game.component.html` file, the CDK generates a `step.stepLabel.template` automatically for each step, which we then use inside the `game-stepper.component.html` file, to show the label of each `<cdk-step>` element respectively. If we didn't provide the `<ng-template cdkStepLabel>` for a step and rather just used `<cdk-step label="someValue">`, it would then use the `step.label` property as per our code written in the `game-stepper.component.html`.

For the bottom navigation buttons, you notice that we use `<button>` elements with the `cdkStepperPrevious` and `cdkStepperNext` directives for going to the previous step and the next step respectively. We also show/hide the next and previous buttons based on the conditions to check if there is a step to go to. To prevent the navigation button from being rendered in the DOM when the provided condition is false, we utilize the `*ngIf` binding to dynamically hide or show it.

One interesting thing about the CDK Stepper API is that we can tell whether the user should be able to go to the next steps and backward, regardless of the state of the current step, or whether the user should first do something in the current step to go to the next one. The way we do it is by using the `[linear]` attribute on our `<app-game-stepper>` element, setting its value to `true`. This tells the CDK Stepper API to not move to the next step using the `cdkStepperNext` button, until the current step's `completed` property is `true`. While just providing `[linear]="true"` is enough to handle the functionality, we improve the UX by disabling the **Next** button—in this case, by using `[disabled]="!selected!.completed"` on the `cdkStepperNext` button, as it makes more sense to just disable the button if it isn't going to do anything on click.

Also, we needed to decide when a step is considered complete. For the first step, it is obvious that we should have a name entered in the input to consider the step completed—or, in other words, the `FormControl` for the `'name'` property in the `nameForm` Form Group should have a value. For the second step, it is bound to the condition `isCorrectGuess !== null`. This ensures that the user has guessed a number; regardless of whether the guess is correct or not, we mark the step as completed and let the user go to the next step (the leaderboard) if the user wants to. And that's pretty much about it. You could go beyond the recipe to add an option of restarting the game from the leaderboard (last) step.

See also

- Angular CDK Stepper examples: `https://material.angular.io/cdk/stepper/examples`

Accessible listbox interactions using CDK Listbox directives

Accessibility is one of the key aspects to focus on when you're thinking about your end users using your applications. One of the major use cases for the users is to make a selection in apps, whether that's selecting the color or size of a product or selecting multiple labels for a post. It makes it such a good experience if the experience is accessible. In this recipe, we're going to replace a color selection feature for products with an accessible alternative using the **Angular CDK Listbox APIs**.

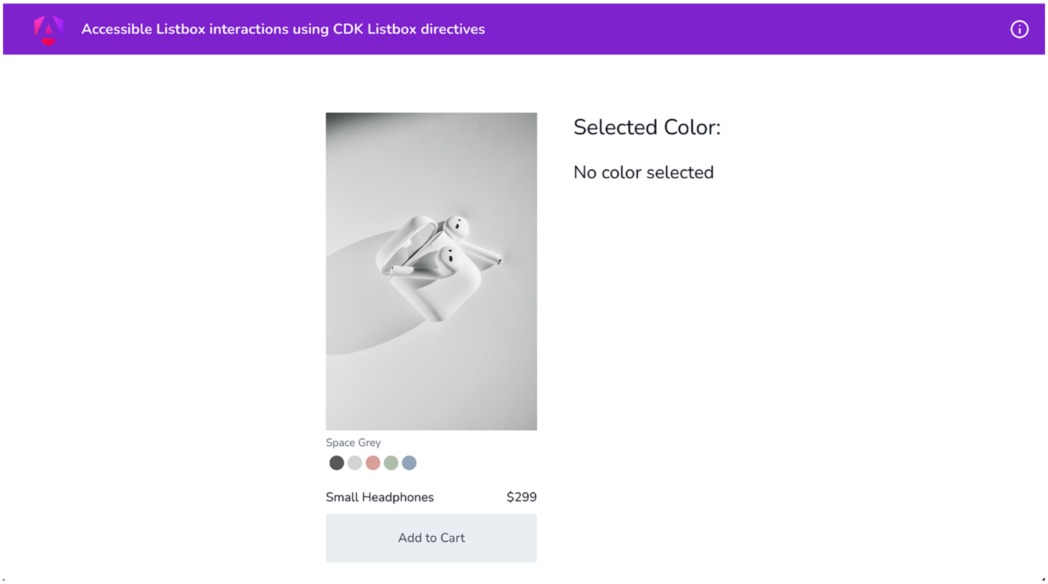
Getting ready

The app that we are going to work with resides in `start/apps/chapter09/ng-cdk-listbox` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-cdk-listbox
```

This should open the app in a new browser tab, and you should see the following:



Accessible Listbox interactions using CDK Listbox directives
Selected Color:
No color selected
Space Grey
Small Headphones
$299
Add to Cart

Figure 9.16: The ng-cdk-listbox app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

In *Figure 9.16*, you will notice that we have options for selecting the product color—and if you click one of them, you can see the selected color on the right side. Even though they are somewhat accessible, they do not completely follow any of the WAI ARIA patterns. Let's use the CDK Listbox module by following these steps:

1. Now, we need to import the `CdkListboxModule` class into the `imports` array of our `ProductCardComponent` class so we can use the `cdkListbox` and `cdkOptions` directives. Modify the `product-card.component.ts` file, as follows:

```
...
import { CdkListboxModule } from '@angular/cdk/listbox';
@Component({
  ...
  imports: [CommonModule, ReactiveFormsModule, CdkListboxModule],
  ...
})
...
```

2. Now, we'll replace the HTML template to use the `cdkListbox` and `cdkOption` directives. We will be using an unordered list (`ul`) and list items (`li` elements) to render a similar color list. Update the `product-card.component.html` file between the mentioned HTML comments as follows:

```
<!-- Color Options -->
        <ul cdkListbox
          [tabindex]="0"
          aria-labelledby="product-colors"
          cdkListboxOrientation="horizontal"
          class="flex flex-wrap justify-center gap-1
           [&:hover_li]:opacity-75">
          <li cdkOption="space-gray" class="color-option
            color-option--space-gray"></li>
          <li cdkOption="silver" class="color-option
            color-option--silver"></li>
          <li cdkOption="pink" class="color-option color-
            option--pink"></li>
          <li cdkOption="green" class="color-option
            color-option--green"></li>
          <li cdkOption="blue" class="color-option color-
            option--blue"></li>
        </ul>
<!-- Color Options end -->
```

You'll notice that the options still look the same (because of the CSS classes); however, the functionality is gone now.

3. Let's also add some styles to highlight the selected color. Update the `color-options.scss` file inside the `src/styles` folder as follows:

```
.color-option {
  ...
  &--active,
  &[aria-selected=true] {
    @apply scale-125;
  }
}
```

4. Since Angular CDK Listbox works with both template-driven and reactive forms, let's add the `formControlName` attribute to bind it with the `color` form control from the `productForm` form group. Update the template further as follows:

```
<!-- Color Options -->
        <ul cdkListbox
          [tabindex]="0"
          formControlName="color" ...>
          ...
        </ul>
<!-- Color Options end -->
```

Great! With this change, when you refresh the page and pick a color, you should be able to see the desired result as before.

5. Angular CDK implements WAI ARIA and, therefore, it is also possible to use the `typeahead` feature in the listbox. Let's modify the template further to make it work:

```
<!-- Color Options -->
        <ul cdkListbox
          ...>
          <li cdkOption="space-gray" cdkOptionTypeaheadLabel="space gray"
class="color-option color-option--space-gray"></li>
          <li cdkOption="silver" cdkOptionTypeaheadLabel="silver"
class="color-option color-option--silver"></li>
          <li cdkOption="pink" cdkOptionTypeaheadLabel="pink"
class="color-option color-option--pink"></li>
          <li cdkOption="green" cdkOptionTypeaheadLabel="green"
class="color-option color-option--green"></li>
          <li cdkOption="blue" cdkOptionTypeaheadLabel="blue"
class="color-option color-option--blue"></li>
        </ul>
<!-- Color Options end -->
```

If you select any color and focus on the list box, you can type the name of the color here to jump directly to the option.

Awesome!! You've finished the recipe. Now you can see the next section to understand how it works.

How it works...

In the recipe, we've used the CDK Listbox API. And it is super cool that it completely implements the WAI ARIA pattern for the listbox. Kudos to the Angular team. To start, we import the `CdkListboxModule` in our standalone component (`ProductCardComponent` class) in its `imports` array. Then we use an unordered list, `<ul>`, in the template along with the `cdkListbox` directive. This applies the CDK Listbox features to the `<ul>` element. You'll notice that we're using the `aria-labelledby` attribute on the `<ul>` according to the WAI ARIA pattern for the listbox. Finally, we have the `cdkListboxOrientation` attribute set to `"horizontal"`, which tells the CDK Listbox API to allow horizontal navigation for this listbox; i.e., we can navigate with the left and right arrow keys. I would encourage you to read the Listbox specification at `https://www.w3.org/WAI/ARIA/apg/patterns/listbox/` to see the cool accessibility features it proposes. And what's amazing is that the Angular CDK Listbox APIs follow all of them.

On top of following the WAI ARIA pattern for listboxes, the listbox itself works seamlessly with Angular forms (both template-driven and reactive). And since we were already using a reactive form in the template of the `product-card` component, we could easily use the `formControlName` binding with the `<ul>` element. Notice that each list item `(<li>)` element has the `cdkOption` binding, which tells the CDK Listbox module about the value of each option. At the end of the recipe, we also introduced the `cdkOptionTypeaheadLabel`, which helps the CDK Listbox API follow the `typeahead` feature of the WAI ARIA pattern recommendation. If we don't use this attribute, the CDK Listbox module uses the list item's text content as the default `typeahead`.

You might have noticed that we also added another CSS selector in the `color-options.scss` file, i.e., `&[aria-selected=true]`, to highlight the selected color. That's because the CDK Listbox API automatically sets that attribute to `true` or `false` based on which item is selected. And this too, is a part of the WAI ARIA pattern for listbox.

See also

- CDK Listbox documentation: `https://material.angular.io/cdk/listbox/overview`
- WAI ARIA pattern for listboxes: `https://www.w3.org/WAI/ARIA/apg/patterns/listbox/`

Working with nested menus using the Angular CDK Menu API

Menus are vital for so many applications we use today. The pattern of having a menu itself allows us to have experiences that don't take up space in the DOM all the time and can be shown relative to their triggers **on demand**. In this recipe, you'll learn how to create nested menus using the Angular CDK APIs.

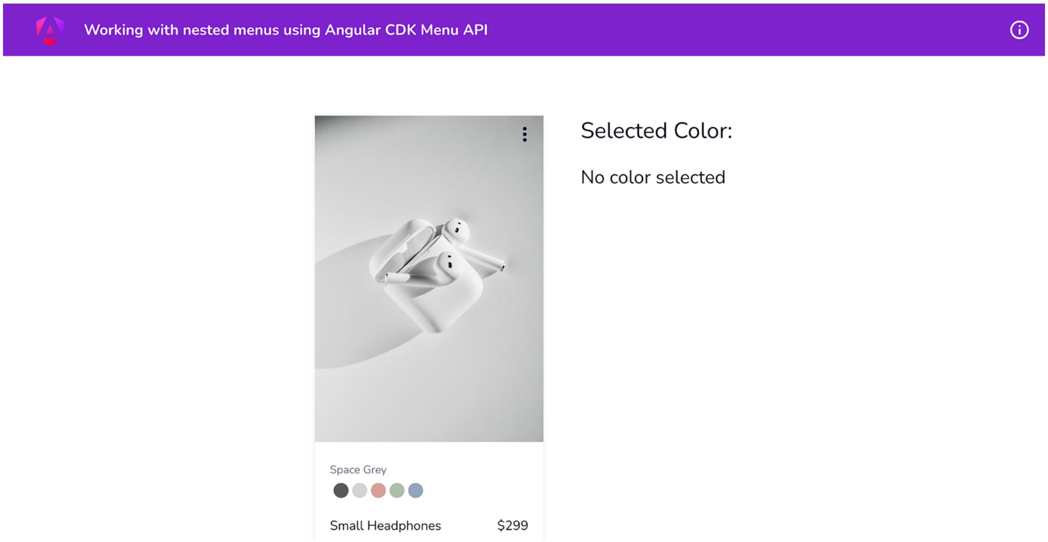
Getting ready

The app that we are going to work with resides in `start/apps/chapter09/ng-cdk-menu` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve ng-cdk-menu
   ```

   This should open the app in a new browser tab, and you should see the following:



Working with nested menus using Angular CDK Menu API

Selected Color:

No color selected

Figure 9.17: The ng-cdk-menu app running on http://localhost:4200

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

We have an application that has a product card that needs an actions menu so we can perform some actions. We'll start using the Angular CDK Menu APIs to implement a nested menu. Let's take the following steps:

1. Copy the styles from `https://ng-cookbook.com/s/cdk-pane-styles` into the `styles.scss` file in the project. Make sure to keep the existing styles in your file and just add the ones you copied.
2. Import the components from the Angular CDK Menu APIs in the `product-card.component.ts` file as follows:

   ```
   ...
   import { CdkMenu, CdkMenuItem, CdkMenuTrigger } from '@angular/cdk/menu';
   @Component({
     ...
   ```

```
  imports: [CommonModule, ReactiveFormsModule, CdkListboxModule ,
CdkMenu, CdkMenuItem, CdkMenuTrigger],
  ...
})
export class ProductCardComponent {...}
```

3. Let's use some directives in the template to create a menu. We'll be using an `<ng-template>` element for our menu and will assign it to a template variable. Update the `product-card.component.html` file as follows:

```
<article class="block group shadow-md bg-white relative">
  <button [cdkMenuTriggerFor]="productActions" class="...">...</button>


  ...
</article>

<ng-template #productActions>
  <ul class="menu-popover__list" cdkMenu>
    <li cdkMenuItem class="menu-popover__list__item">
      Share
    </li>
    <li cdkMenuItem class="menu-popover__list__item">
      Manage &#10148;
    </li>
    <li cdkMenuItem class="menu-popover__list__item">
      Preview
    </li>
  </ul>
</ng-template>
```

If you click the More button (three vertical dots icon) on the product card now, you should be able to see the menu as follows:

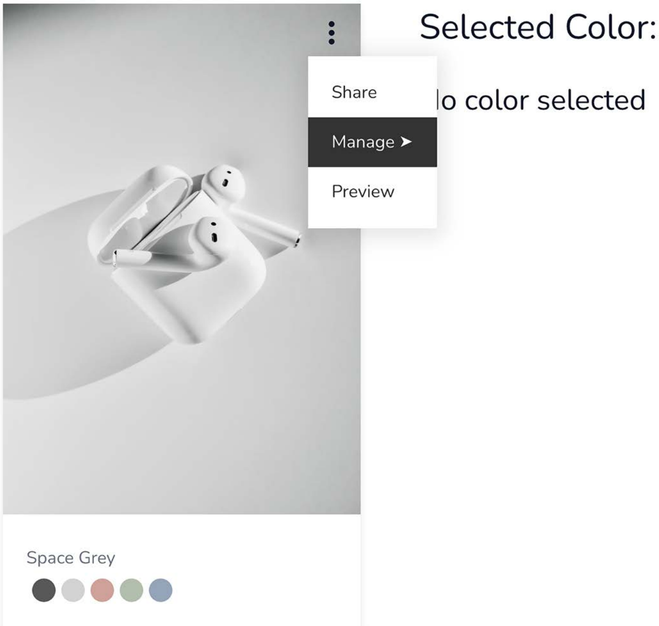


Figure 9.18: Menu implementation using Angular CDK Menu APIs

You'll notice that the menu starts from the bottom-left position of the trigger. This is the default menu position set by the Angular CDK Menu API. We can override it using connected positions.

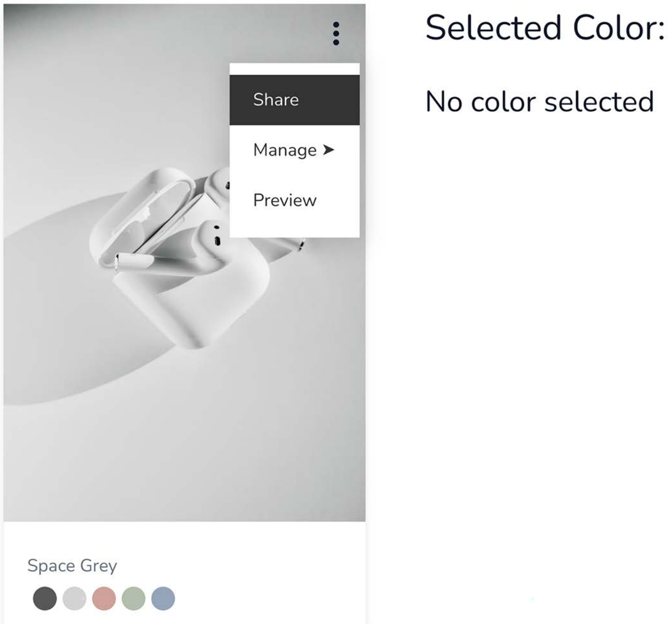
4. Let's use some preferred connected positions so the menu is shown on the left side of the trigger by default. Update the `product-card.component.ts` file as follows:

```
...
import { ConnectedPosition } from '@angular/cdk/overlay';
...
export class ProductCardComponent {
  ...
  menuPositions: ConnectedPosition[] = [
    {
      offsetY: 4,
      originX: 'end',
        originY: 'bottom',
        overlayX: 'end',
        overlayY: 'top',
      },
      {
        offsetY: -4,
        originX: 'end',
        originY: 'top',
        overlayX: 'end',
        overlayY: 'bottom',
      },
    ]
  }
```

5. Let's use the `menuPositions` variable in the template to use the preferred positions. Update the `product-card.component.html` file as follows:

```
<article class="block group shadow-md bg-white relative">
  <button [cdkMenuPosition]="menuPositions"
    [cdkMenuTriggerFor]="productActions" class="...">
    <span class="material-symbols-outlined">
      more_vert
    </span>
  </button>
</article>
```

If you now refresh the app and click the More button (three vertical dots icon) on the product card, you should see the menu appearing on the left side of the button as follows:



Share
Manage ➤
Preview
Selected Color:
No color selected
Space Grey

Figure 9.19: Menu with preferred positions

6. Let's implement the nested menu now. We'll create the `<ng-template>` element for the nested menu and connect it with the `"Manage"` menu item. Update the `product-card.component.html` file as follows:

```
<article class="block group shadow-md bg-white
  relative">...</article>
<ng-template #productActions>
  <ul class="menu-popover__list" cdkMenu>
    <li cdkMenuItem class="menu-popover__list__item">...</li>
    <li cdkMenuItem class="menu-popover__list__item"
      [cdkMenuTriggerFor]="productEditActions">
      Manage &#10148;
    </li>
    <li cdkMenuItem class="menu-
      popover__list__item">...</li>
  </ul>
</ng-template>
```

```
<ng-template #productEditActions>
  <ul class="menu-popover__list" cdkMenu>
    <li cdkMenuItem class="menu-popover__list__item">
      Duplicate
    </li>
    <li cdkMenuItem class="menu-popover__list__item">
      Edit
    </li>
    <li cdkMenuItem class="menu-popover__list__item">
      Delete
    </li>
  </ul>
</ng-template>
```

With this change, you should be able to see the nested menu as follows:

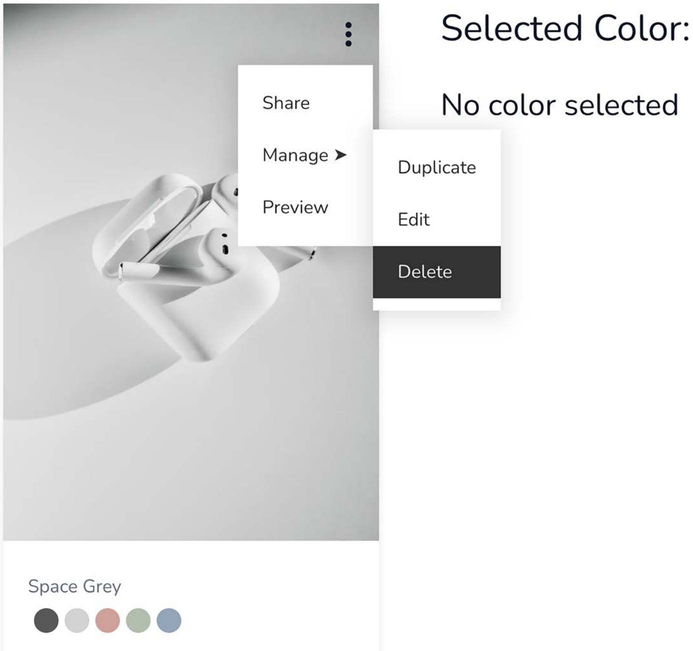

Share
Manage ➤
Preview
Duplicate
Edit
Delete
Selected Color:
No color selected
Space Grey

Figure 9.20: Nested menu implementation

Kaboom! We were able to implement a nested menu in this application within a few steps by using the Angular CDK. Trust me, if we had to do this on our own, this would've taken hours, or maybe days.

How it works...

The Angular CDK provides the CDK Menu APIs, which include the `cdkMenuTriggerFor` directive. This directive ideally points to a template variable for an `<ng-template>` element. And then behind the scenes, the Angular CDK connects that menu with the trigger. Even though the menu item itself is appended to the `<body>` element inside an overlay, the position of the overlay is calculated accurately for the menu and is correct across screen resizes as well. However, the Angular CDK Menu API **does not** contain the CSS styles for the overlay. Therefore, in this recipe, we have a link to the relevant styles for both the overlay and the menu itself. You'll also notice that we use the directives `cdkMenu` on the menu list `(<ul>)` items, and we use the `cdkMenuItem` on the menu `(<li>)` items. Since we use these three directives and nothing else from the Angular CDK Menu package, you'll notice that we don't import `CdkMenuModule` from `'@angular/cdk/menu'`. That's because all these directives are `standalone` directives. So they can be imported without the module.

In this recipe, we also modified the default positioning for the menu. By default, the overlay starts from the *bottom-left* point of the trigger, toward the right side. We used a `ConnectionPosition` array with the `[cdkMenuPosition]` binding to the **More** button (the trigger element) to use our preferred position. The first (preferred) position is that the overlay's **top-right** corner aligns with the trigger element's **bottom-right** corner. That is, the overlay shows below the **More** button and expands toward the left side, as per the following object:

```
{
      offsetY: 4,
      originX: 'end',
      originY: 'bottom',
      overlayX: 'end',
      overlayY: 'top',
 }
```

The second position is that the overlay shows above More button (three vertical dots icon) and still expands toward the left side based on the following options:

```
{
      offsetY: -4,
      originX: 'end',
      originY: top,
      overlayX: 'end',
      overlayY: bottom,
 }
```

Finally, we implemented the nested menu. This was pretty simple. We created another menu with the same technique as the first one, creating an `<ng-template>` with a template variable, a `cdkMenu` inside it, and `cdkMenuItems` inside the menu. Then we set the trigger of this nested menu to the `Manage` menu item in the primary menu. The Angular CDK is smart enough to understand if the trigger is a `cdkMenuItem`. If it is, it opens the nested menu automatically on the mouse-over event on the trigger menu item. And when the mouse-leave event triggers from the nested menu, it hides. I must give kudos to the Angular CDK team for making things this easy.

See also

- Angular CDK Menu: `https://material.angular.io/cdk/menu/overview`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

10

Writing Unit Tests in Angular with Jest

"It works on my machine" is a phrase that won't lose its beauty with time. It is a shield for many engineers and a nightmare for the quality assurance professionals. But honestly, what's a better way for your application's robustness than writing tests? And when it comes to writing unit tests, my personal favorite is **Jest**. That is because it is super-fast, lightweight, and has an easy API to write tests. More importantly, it is faster than the Karma and Jasmine setup that comes out of the box with Angular. In this chapter, you'll learn how to configure Jest with Angular to run these tests in parallel. You'll learn how to test components, services, and pipes with Jest. You'll also learn how to mock dependencies for these tests.

In this chapter, we're going to cover the following recipes:

- Setting up unit tests in Angular with Jest
- Providing global mocks for Jest
- Mocking Angular services using stubs
- Using spies on an injected service in a unit test
- Mocking child components and directives using the `ng-mocks` package
- Writing even easier tests with Angular CDK component harnesses
- Unit-testing components with observables
- Unit-testing Angular pipes

Technical requirements

For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter10`.

Setting up unit tests in Angular with Jest

By default, a new Angular project comes bundled with a lot of goodness, including the configuration and tooling in which to run unit tests with Karma and Jasmine. While working with Karma is relatively convenient, many developers find that in large-scale projects, the whole testing process becomes much slower if there are a lot of tests involved. This is mainly because you can't run tests in parallel. In this recipe, we'll set up Jest for unit testing in an Angular app. Additionally, we'll migrate existing tests from the Karma syntax to the Jest syntax.

> Starting with v16, Angular comes with a developer preview for Jest, and the process has become much easier. This recipe targets apps up to Angular v15 and covers what we can do in v16 at the end of the chapter.

Getting ready

The app that we are going to work with now resides in `start/apps/chapter10/ng-jest-setup` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve:ng-jest-setup
```

> The command in *step 2* is different from the rest of the recipes' commands in this book. This is because the project for this recipe is *not* part of the NX workspace we have. It is a standalone Angular app itself; hence, it has different commands.

This should open the app in a new browser tab, and you should see the following:

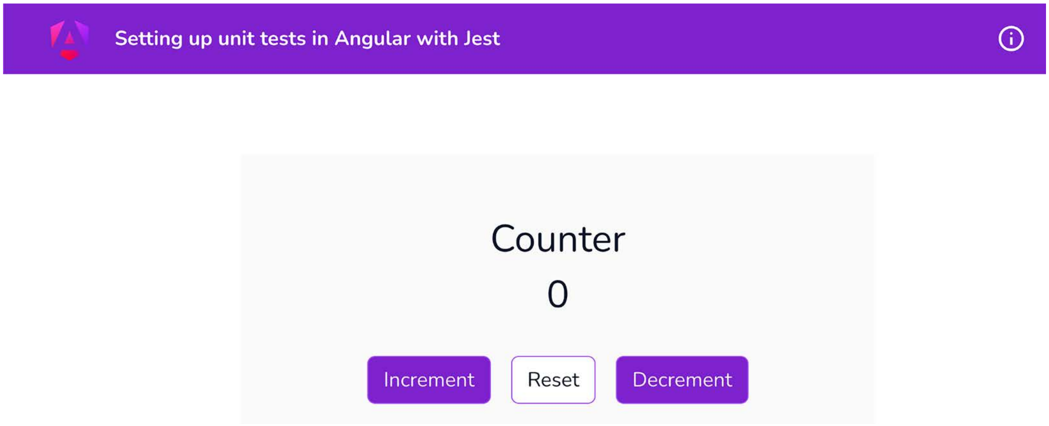
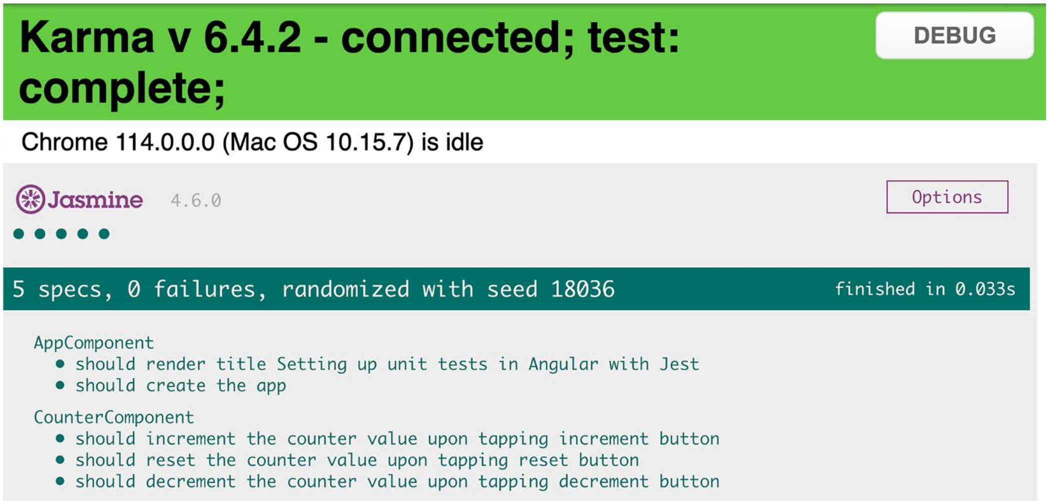


Figure 10.1: The ng-jest-setup app running on http://localhost:4200

Next, try to run the test and monitor how much time it takes for the entire process to run. Run the command `npm run test:ng-jest-setup` from the workspace `root`; within a few seconds, a new Chrome window should open, as follows:



Karma v 6.4.2 - connected; test: complete;

Chrome 114.0.0.0 (Mac OS 10.15.7) is idle

```
5 specs, 0 failures, randomized with seed 18036                 finished in 0.033s

AppComponent
  • should render title Setting up unit tests in Angular with Jest
  • should create the app
CounterComponent
  • should increment the counter value upon tapping increment button
  • should reset the counter value upon tapping reset button
  • should decrement the counter value upon tapping decrement button
```

Figure 10.2: The tests' results with Karma and Jasmine

Looking at the preceding screenshot, you might say *"Pfffttt, Ahsan, it says 'finished in 0.033s (or something close to it)!' What else do you want?"* Well, that time only covers how long it took Karma to run the tests *in the browser after the Chrome window was created*. It doesn't count the time it took to start the process, start the Chrome window, and then load the tests. This could be improved by running the Chrome Headless browser for Karma, which requires some configuration. However, it is still slower compared to the alternative (Jest). Also, at the time of writing this recipe, I'm running this on a Macbook Pro with the M2 Chip, *which is fast*. That's why we're going to replace it with Jest. Now that you understand the issue, in the next section, let's take a look at the steps of the recipe.

How to do it...

Here, we have an Angular app with a simple `Counter` component. It shows the value of the counter and has three action buttons: one of the buttons is to increase the value of the counter, one of them is to decrease the value, and another is to reset the value. Additionally, there are some tests written with Karma and Jasmine, and all of the tests will pass if you run the tests. We'll start by setting up Jest instead. Perform the following steps:

1. First, open a new Terminal window/tab, and make sure you're inside the `start/apps/chapter10/ng-jest-setup` folder. Once inside, run the following command to install the packages that are required to test with Jest:

```
npm install --save-dev jest jest-preset-angular @types/jest
```

2. We can now uninstall Karma and the unwanted dependencies. Now, run the following command in your Terminal:

```
npm uninstall karma karma-chrome-launcher karma-jasmine-html-reporter @
types/jasmine @types/jasminewd2 jasmine-core jasmine-spec-reporter karma-
coverage-istanbul-reporter karma-jasmine
```

3. Now, update the test configuration in the `angular.json` file, as follows:

```
{
  ...
  "projects": {
    "ng-jest-setup": {
      "..."
      "prefix": "app",
      "architect": {
        "build": {...},
        "serve": {...},
        "extract-i18n": {...},
        "test": {
          "builder": "@angular-builders/jest:run",
          "options": {
            "tsConfig":
              "<rootDir>/src/tsconfig.test.json",
            "collectCoverage": false,
            "forceExit": true
          }
        },
        "lint": {...},
        "e2e": {...}
      }
    }
  },
  "defaultProject": "setting-up-jest"
}
```

4. We will now create a file to configure Jest for our project. Create a file named `jestSetup.ts` inside the project's folder, and paste the following content inside:

```
import 'jest-preset-angular/setup-jest';
```

5. Now, let's modify `tsconfig.spec.json` to use Jest instead of Jasmine. After the modification, your entire file should appear as follows:

```
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "outDir": "./out-tsc/spec",
    "types": ["jest", "node"],
    "esModuleInterop": true,
    "emitDecoratorMetadata": true
  },
  "files": ["src/polyfills.ts"],
  "include": ["src/**/*.spec.ts", "src/**/*.d.ts"]
}
```

6. We'll now modify `package.json` to add the `npm` scripts that'll run the Jest tests:

```
{
  "name": "setting-up-jest",
  "version": "0.0.0",
  "scripts": {
    ...
    "build": "ng build",
    "test": "jest",
    "test:coverage": "jest --coverage",
    ...
  },
  "private": true,
  "dependencies": {...},
  "devDependencies": {...},
}
```

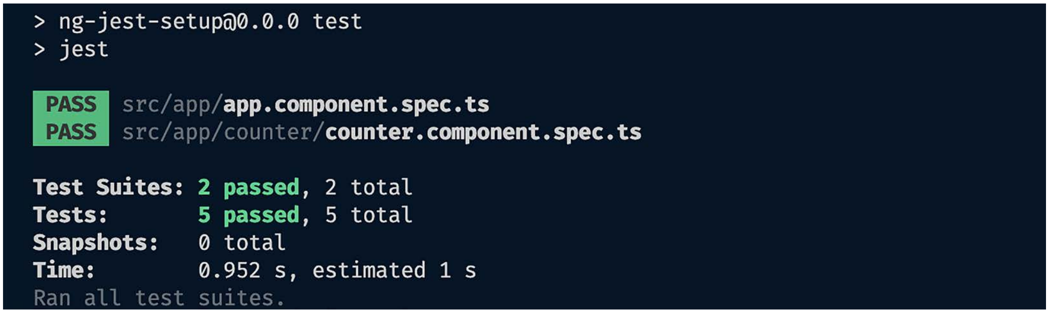
7. Finally, let's wrap up the entire configuration for our Jest tests by adding the Jest configuration in the `package.json` file, as follows:

```
{
  ...
  "dependencies": {...},
  "devDependencies": {...},
  "jest": {
    "preset": "jest-preset-angular",
    "setupFilesAfterEnv": [
      "<rootDir>/jestSetup.ts"
    ]
  }
}
```

8. Now that we have set everything up, simply run the `test` command from within the `ng-jest-setup` folder, as follows:

   ```
   npm run test
   ```

   Once the tests are finished, you should be able to see the following output:



```
> ng-jest-setup@0.0.0 test
> jest

 PASS  src/app/app.component.spec.ts
 PASS  src/app/counter/counter.component.spec.ts

Test Suites: 2 passed, 2 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        0.952 s, estimated 1 s
Ran all test suites.
```

Figure 10.3: The results of the tests with Jest

Kaboom! You will notice that the entire process of running the tests with Jest takes about 6 seconds. It might take longer when you run it for the first time, but the subsequent runs should be faster. Now that you know how to configure an Angular app to use Jest for unit tests, please refer to the next section for resources in which to learn more.

Bonus: Migrating to Jest in Angular v16

Starting from Angular v16, all you need to do to migrate to Jest (considering your project currently uses Karma) is to update the `angular.json` file in the project with the following configuration object property hierarchy `your-app > architect > test` settings:

```
{
  "projects": {
    "my-app": {
      "architect": {
        "test": {
          "builder": "@angular-devkit/build-angular:jest",
          "options": {
            "tsConfig": "tsconfig.spec.json",
            "polyfills": ["zone.js", "zone.js/testing"]
          }
        }
      }
    }
  }
}
```

See also

- Testing Web Frameworks with Jest: `https://jestjs.io/docs/en/testing-frameworks`
- Getting Started with Jest: `https://jestjs.io/docs/en/getting-started`
- Moving Angular CLI to Jest: `https://blog.angular.io/moving-angular-cli-to-jest-and-web-test-runner-ef85ef69ceca`

Providing global mocks for Jest

In the previous recipe, we learned how to set up Jest for Angular unit tests. There might be some scenarios in which you'd want to use a browser API that might not be part of your actual Angular code – for instance, using `LocalStorage` or `alert`. In such cases, we need to provide some global mocks for the functions that we want to return mock values from. This is so that we can perform tests involving them as well. In this recipe, you'll learn how to provide global mocks to Jest.

Getting ready

The app that we are going to work with now resides in `start/apps/chapter10/ng-jest-global-mocks` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-jest-global-mocks
```

This should open the app in a new browser tab, and you should see the following:

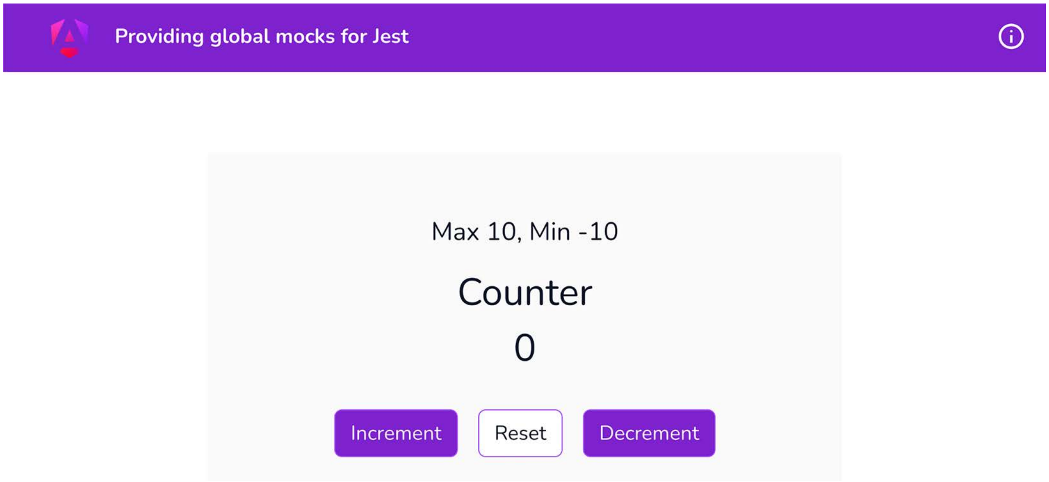


Figure 10.4: The ng-jest-global-mocks app running on http://localhost:4200

Now that we have the app running locally, in the next section, let's go through the steps of the recipe.

How to do it...

The app we're using for this recipe uses two global APIs: `window.localStorage` and `window.alert`. Note that when the app starts, we get the counter value from `LocalStorage`, and then upon increment, decrement, and reset, we store it in `LocalStorage`. When the counter value becomes greater than the `MAX_VALUE` or lower than the `MIN_VALUE`, we show the alert using the `alert` method. Let's begin the recipe by writing some cool unit tests:

1. First, we'll write our test cases to show the alert when the counter value goes beyond `MAX_VALUE` and `MIN_VALUE`. Modify the `counter.component.spec.ts` file as follows:

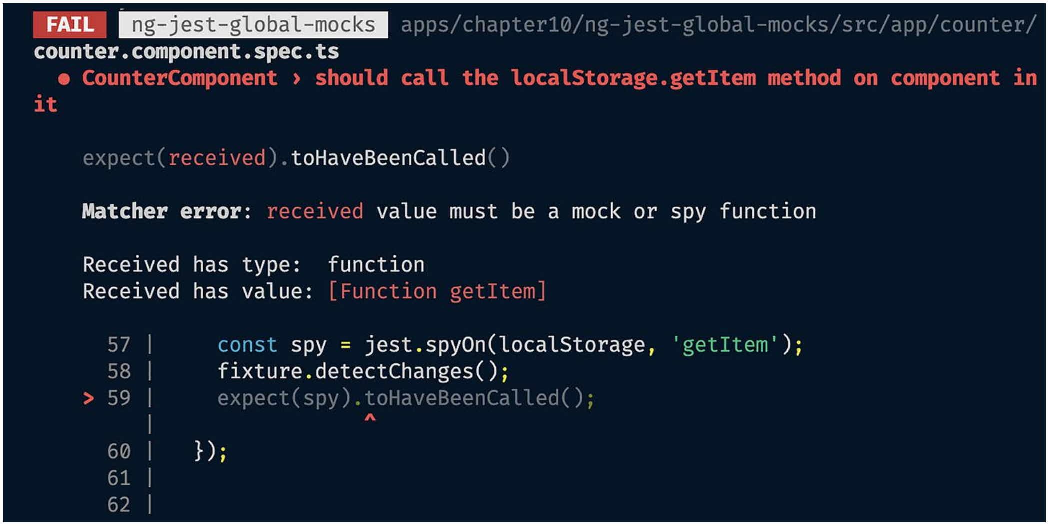
```
...
describe('CounterComponent', () => {
  ...
  it('should show an alert when the counter value goes
    above the MAX_VALUE', () => {
    jest.spyOn(window, 'alert');
    component.counter = component.MAX_VALUE;
    component.increment();
    expect(window.alert).toHaveBeenCalledWith('Value too
      high');
    expect(component.counter).toBe(component.MAX_VALUE);
  });
  it('should show an alert when the counter value goes
    below the MIN_VALUE', () => {
    jest.spyOn(window, 'alert');
    component.counter = component.MIN_VALUE;
    component.decrement();
    expect(window.alert).toHaveBeenCalledWith('Value too
      low');
    expect(component.counter).toBe(component.MIN_VALUE);
  });
});
```

Here, you can see that the tests pass but with a bunch of console errors. And what if we wanted to check whether the value from `LocalStorage` is being saved and retrieved properly?

2. We'll create a new test to make sure the `localStorage.getItem` method is called to retrieve the last saved value from the `LocalStorageAPI`. Add the test to the `counter.component.spec.ts` file, as follows:

```
...
describe('CounterComponent', () => {
  ...
  it.only('should call the localStorage.getItem method on
    component init', () => {
    jest.spyOn(localStorage, 'getItem');
    component.ngOnInit();
    expect(localStorage.getItem).toHaveBeenCalled();
  });
});
```

Notice that we're using `it.only` for this test case. This is to ensure that we're only running this test (for now). If you run the tests, you should be able to see something like the following screenshot:



```
 FAIL   ng-jest-global-mocks  apps/chapter10/ng-jest-global-mocks/src/app/counter/
counter.component.spec.ts
  ● CounterComponent › should call the localStorage.getItem method on component in
it

    expect(received).toHaveBeenCalled()

    Matcher error: received value must be a mock or spy function

    Received has type:  function
    Received has value: [Function getItem]

      57 |     const spy = jest.spyOn(localStorage, 'getItem');
      58 |     fixture.detectChanges();
    > 59 |     expect(spy).toHaveBeenCalled();
         |                 ^
      60 |   });
      61 |
      62 |
```

Figure 10.5: The test that covers the LocalStorageAPI has failed

Notice the **Matcher error: received value must be a mock or a spy function** message. This is what we're going to do next, that is, provide a mock.

3. Create a file inside the project's `src` folder, and name the file `jest-global-mocks.ts`. Then, add the following code to mock the `LocalStorageAPI`:

```
class LocalStorageMock {
  storage: Partial<Storage> = {};
  getItem(key: string) {
    return this.storage[key] ? this.storage[key] : null;
  }

  setItem(key: string, value: string) {
    this.storage[key] = value;
  }
}
```

```
Object.defineProperty(window, 'localStorage', {
  value: new LocalStorageMock(),
});

Object.defineProperty(window, 'alert', {
  value: jest.fn(),
});
```

4. Now, import this file into the `src/test-setup.ts` file, as follows:

```
...
globalThis.ngJest = {...};
import 'jest-preset-angular/setup-jest';
import './jest-global-mocks';
```

   Now, if you rerun the tests, they should pass.

5. Let's add another test to ensure we retrieve the last saved value from `LocalStorage` in the component initiation. Modify the `counter.component.spec.ts` file, as follows:

```
...
describe('CounterComponent', () => {
  ...
  it('should call the localStorage.getItem method on
    component init', () => {...});
  it('should retrieve the last saved value from
    localStorage on component init', () => {
    localStorage.setItem('counterValue', '12');
    component.ngOnInit();
    expect(component.counter).toBe(12);
  });
});
```

6. Finally, let's make sure that we save the counter value to `LocalStorage` whenever we trigger the `increment`, `decrement`, or `reset` methods. Update the `counter.component.spec.ts` as follows:

```
...
describe('CounterComponent', () => {
  ...
  it('should save the new counterValue to localStorage on
    increment, decrement and reset', () => {
    jest.spyOn(localStorage, 'setItem');
    component.counter = 0;
    component.increment();
    expect(localStorage.setItem).toHaveBeenCalledWith(
      'counterValue', '1');
    component.counter = 20;
    component.decrement();
    expect(localStorage.setItem).toHaveBeenCalledWith(
      'counterValue', '19');
    component.reset();
    expect(localStorage.setItem).toHaveBeenCalledWith(
      'counterValue', '0');
  });
});
```

Awesomesauce! You've just learned how to provide global mocks to Jest for testing. Please refer to the next section to understand how this works.

How it works...

Jest provides a way to define global mocks for tests. It comes with a lot of built-in functions to define test suites and test cases, along with assertion functions. Here is an example of a simple Jest test suite for a `sum` function that adds two numbers:

```
import { sum } from './sum';
describe('Sum function', () => {
  it('should add two numbers', () => {
    expect(sum(2,3)).toBe(5);
  });
});
```

Jest also has a way to use a particular file to set up Jest in projects. When working with different frameworks, the file (or the filename) can be different, but the purpose would still be the same, that is, to define global configuration via this Jest setup file. In our NX workspace, we already have the Jest files in each project under the `src` folder, and the filename is usually `test-setup.ts`. One of the common approaches to define global mocks is to create a new file containing the mocks and then import it into the `test-setup.ts` file.

Notice that we use the `Object.defineProperty` method in the `window` object to provide a mock implementation for the `LocalStorage` object, and we do similar for the `window.alert` mock. This is actually the same for any API that is not implemented in the **JSDOM(JavaScript Document Object Model**). Similarly, you can provide a global mock for each API that you use in your tests. Notice that in the `value` property, we create a new instance of the `LocalStorageMock` class using the `new` keyword. Essentially, this is one way to define mocks. We create the `LocalStorageMock` class, and in that, we have a private property named `storage` that mimics the `localStorage` object. We have also defined the `getItem` and `setItem` methods in there so that we can set values to this storage and get values from it. Notice that we do not have the implementations of the `removeItem` and `clear` methods that we have in the actual `localStorage` API. We don't have to have them because we're not using these methods in our actual code.

In the `should call the localStorage.getItem method on component init` test, we simply spy on the `localStorage` object's `getItem` method, call the `ngOnInit` method ourselves, and then expect it to have been called. *Easy peasy!*

In the `should retrieve the last saved value from localStorage on component init` test, we save a value in the `localStorage` object for the counter value as `12`, using the `setItem` method. Essentially, calling the `setItem` method calls our mock implementation method and not the actual `localStorage` API's `setItem` method. Notice that, here, we *do not* spy on the `getItem` method; this is because, later on, we want the value of the component's `counter` property to be `12`.

IMPORTANT NOTE

Whenever we spy on a method, remember that any statements in the actual function will no longer be executed. This is why we do not spy on the `getItem` method in the preceding test. If we do so, the `getItem` method from the mock implementation will not return anything. Therefore, our expected value for the counter property will not be `12`.

Put simply, if you have to rely on the outcome of a function's implementation, or the statements executed within a function, do not spy on that function and replace such a function with `jest.fn().mockReturnValue(123);`. This ensures we can check if the function is called but also with a return value that can be used by the actual code.

PS: I always end up learning this the hard way after debugging and bashing my head for a while.

The final test is an easy one. In the `should save the new counterValue to localStorage on increment, decrement and reset` test, we spy on the `setItem` method. Then, we manually set the value of the counter property each time before we run the `increment`, `decrement`, and `reset` methods respectively. Additionally, we expect the `setItem` method to have been called with the right arguments to save the value to the store. Note that we do not check the store's value after saving it. As I mentioned earlier, since we have spied on the `setItem` method, its internal statement won't trigger and the value won't be saved; therefore, we can't retrieve the saved value afterward.

See also

- The Jest documentation for `setupFiles`: `https://jestjs.io/docs/en/configuration#setupfiles-array`
- Manual Mocks with Jest: `https://jestjs.io/docs/en/manual-mocks`

Mocking Angular services using stubs

There's rarely an Angular app that doesn't have a **Service** created inside it. And where overall business logic is concerned, services hold a great deal, particularly when it comes to interacting with APIs. In this recipe, you'll learn how to mock services using stubs.

Getting ready

The app that we are going to work with now resides in `start/apps/chapter10/ng-test-services-stubs` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-test-services-stubs
```

This should open the app in a new browser tab, and you should see the following:

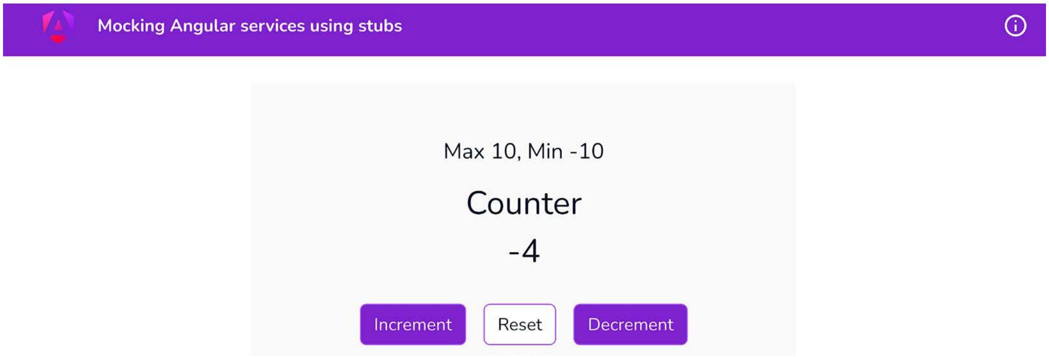


Figure 10.6: The ng-test-services-stubs app running on http://localhost:4200

Now that we have the app running locally, in the next section, let's take a look at the steps of the recipe.

How to do it…

We have the same application as the previous recipe; however, we've moved the logic of saving and retrieving data from `localStorage` to the `CounterService` we've created. Now, all the tests are passing. However, what if we wanted to hide/encapsulate the logic of where the counter value is stored? Perhaps we want to send a backend API call for it. To do this, it makes more sense to test that our `CounterService`'s methods are being called instead of checking `localStorage` methods. Let's follow the recipe to provide a mock stub for our service:

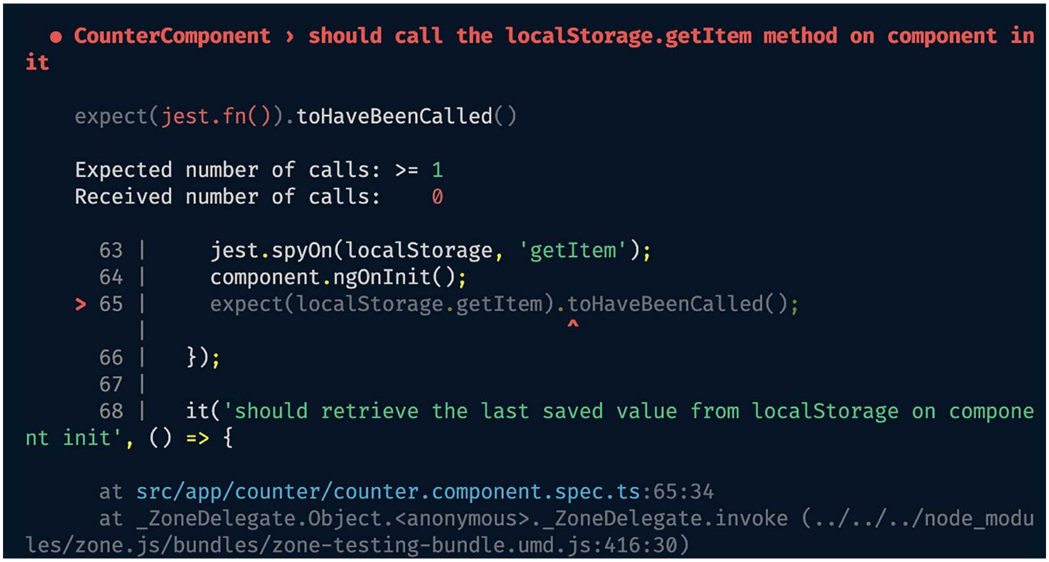
1. First, let's create a folder inside the `src` folder, named `__mocks__`. Inside it, create another folder named `services`. Then, again inside this folder, create the `counter.service.mock.ts` file with the following content:

```
import { CounterService } from "../../app/services/counter.service";
const counterServiceMock: CounterService = {
  storageKey: 'counterValue',
  getFromStorage: jest.fn(),
  saveToStorage: jest.fn(),
};
export default counterServiceMock;
```

2. Now, provide the mock service instead of the actual service in the `counter.component.spec.ts`, as follows:

```
...
import { CounterService } from '../services/counter.service';
import counterServiceMock from '../../__mocks__/services/counter.service.
mock';

describe('CounterComponent', () => {
  ...
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [CounterComponent],
      providers: [{
        provide: CounterService,
        useValue: counterServiceMock
      }]
    }).compileComponents();
  });
  ...
});
```

With the preceding change, you should see the following error that says the `localStorage.getItem` hasn't been called. This is because we're now spying on the methods on our mock stub for the service:



```
  ● CounterComponent › should call the localStorage.getItem method on component in
it

    expect(jest.fn()).toHaveBeenCalled()

    Expected number of calls: >= 1
    Received number of calls:    0

      63 |     jest.spyOn(localStorage, 'getItem');
      64 |     component.ngOnInit();
    > 65 |     expect(localStorage.getItem).toHaveBeenCalled();
         |                                  ^
      66 |   });
      67 |
      68 |   it('should retrieve the last saved value from localStorage on compone
nt init', () => {

      at src/app/counter/counter.component.spec.ts:65:34
      at _ZoneDelegate.Object.<anonymous>._ZoneDelegate.invoke (../../../node_modu
les/zone.js/bundles/zone-testing-bundle.umd.js:416:30)
```

Figure 10.7: localStorage.getItem is not called because of the methods being spied on

3. Now, instead of expecting the `localStorage` object's methods to be called, let's expect our service's methods to be called in our tests. Update the `counter.component.spec.ts` file using the snippet at `https://ng-cookbook.com/s/services-stub-tests` to replace all the tests below the following comment:

   ```
   ...
   describe('CounterComponent', () => {
     ...
     // replace the tests below
   });
   ```

Great! You now know how to mock services to test components with service dependencies. Please refer to the next section to understand how it all works.

How it works...

Providing stubs for Angular services is already a breeze. This is thanks to Angular's out-of-the-box methods and tooling from the `@angular/core` package, especially `@angular/core/testing`. First, we create the stub for our `CounterService` and use `jest.fn` for every method within `CounterService`.

Calling `jest.fn` returns a new, unused mock function that Jest automatically spies upon as well. Optionally, we can also pass a mock implementation method as a parameter for `jest.fn`. View the following example from the official documentation for `jest.fn`:

```
const mockFn = jest.fn();
mockFn();
expect(mockFn).toHaveBeenCalled(); // test passes
// With a mock implementation:
const returnsTrue = jest.fn(() => true);
console.log(returnsTrue()); // true;
expect(returnsTrue()).toBe(true); // test passes
```

Once we create the stub, we pass it to the `TestBed` configuration in the provider's array against the `CounterService` – but with the `useValue` property set to the `counterServiceMock`. This tells Angular to use our stub as it is for `CounterService`.

Then, in the test where we expect `CounterService.getFromStorage` method to be called when the component initiates, we use the following statement:

```
expect(counterServiceMock.getFromStorage).toBeCalled();
```

Notice that in the preceding code, we are able to directly use `expect` on `counterServiceMock.getFromStorage`. While this isn't possible in Karma and Jasmine, it is possible with Jest, since we're using `jest.fn` for each underlying method.

Then, for a test in which we want to check whether the `getFromStorage` method is called and returns a saved value, we first use the `counterServiceMock.getFromStorage.mockReturnValue(12);` statement. This ensures that when the `getFromStorage` method is called, it returns the value of `12`. Then, we just run the `ngOnInit` method in the test and expect that our component's counter property has now been set to `12`. This means that the following things happen:

1. `ngOnInit` calls the `getFromStorage` method.
2. `getFromStorage` returns the previously saved value (in our case, that's `12`, but in reality, it'll be fetched from `localStorage`).
3. The component's counter property is set to the retrieved value, which, in our case, is `12`.

Now, for the final test, we just expect that the `saveToStorage` method of our `CounterService` is called in each necessary case. For this, we use the following types of `expect` statements:

```
expect(counterServiceMock.saveToStorage).toHaveBeenCalledWith(1);
```

That's pretty much about it. Unit tests are fun, aren't they? Now that you've understood how it all works, please refer to the next section for some helpful resources that you can use for further reading.

See also

- The official documentation for `jest.fn`: `https://jestjs.io/docs/en/jest-object.html#jestfnimplementation`
- Angular's *Component testing scenarios*: `https://angular.io/guide/testing-components-scenarios`

Using spies on an injected service in a unit test

While you can provide stubs for your services in the unit tests with Jest, sometimes, it might feel like an overhead creating a mock for every new service. Let's suppose that if the service's usage is limited to one test file, it might make more sense to just use spies on the actual injected service. In this recipe, that's exactly what we're going to do.

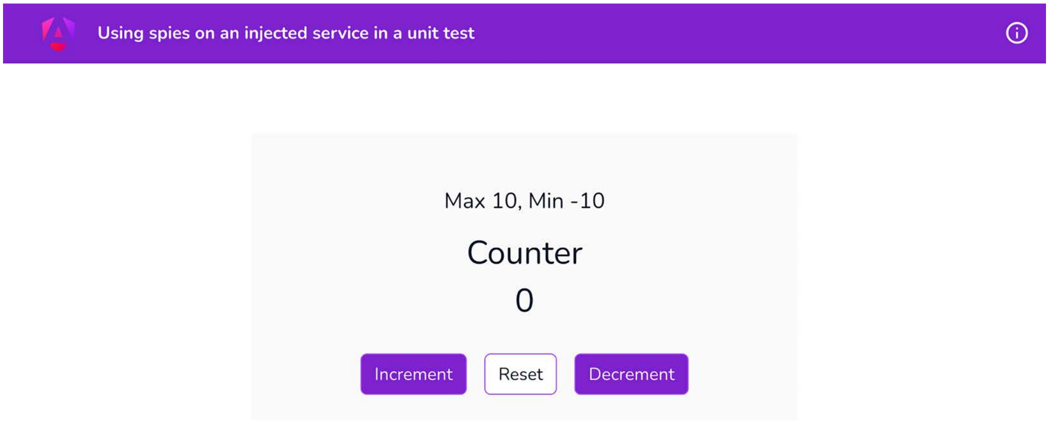
Getting ready

The app that we are going to work with now resides in `start/apps/chapter10/ng-test-services` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve ng-test-services
   ```

This should open the app in a new browser tab, and you should see the following:



Using spies on an injected service in a unit test

Max 10, Min -10

Counter

0

Figure 10.8: ng-test-services app running on http://localhost:4200

Now that we have the app running, in the next section, let's go through the steps of the recipe.

How to do it...

We have an application that has a `Counter` component. We can increment, decrement, and reset the counter, and the value for the counter changes. The value is also stored in the `localStorage`. We have the logic of saving and retrieving data from `localStorage` to the `CounterService`. Now, all the tests are passing, but we are checking the values in our tests directly from the `localStorage`. What if we decided to change the logic of where the counter value is stored? Perhaps we want to send a backend API call for it. To do this, it makes more sense to test that our `CounterService`'s methods are being called instead of checking `localStorage` methods. However, unlike the previous recipe, where we provided stubs for our services, we're going to inject the service directly and will create jest spies for the functions from the Angular Service that we want to ensure are being called. Let's follow the recipe to do that:

1. Open the `counter.component.spec.ts` file. You'll see a comment that says `// replace the tests below`. Let's replace the first test `should call the localStorage.getItem method on component init` below that comment with the following:

```
it(' should get counter from storage on component init',
  () => {
    jest.spyOn(component.counterService, 'getFromStorage');
    component.ngOnInit();
    expect(component.counterService.getFromStorage)
      .toHaveBeenCalled();
  });
```

   If you run `npm run test` again, you should still see all the tests passing.

2. Now, let's replace the test that says `should retrieve the last saved value from localStorage on component init` with the following test:

```
it(' should have the initial value of counter from storage
  on component init ', () => {
    jest.spyOn(component.counterService,
      'getFromStorage').mockReturnValue(12);
    component.ngOnInit();
    expect(component.counter).toBe(12);
  });
```

3. Finally, replace the last test that says `should save the new counterValue to localStorage on increment, decrement and reset` with the following code:

```
it('should save the new counterValue on increment, decrement
  and reset', () => {
    jest.spyOn(component.counterService, 'saveToStorage');
    component.counter = 0;
    component.increment();
    expect(component.counterService.saveToStorage)
      .toHaveBeenCalledWith(1);
    component.counter = 20;
    component.decrement();
    expect(component.counterService.saveToStorage)
      .toHaveBeenCalledWith(19);
    component.reset();
    expect(component.counterService.saveToStorage)
      .toHaveBeenCalledWith(0);
  });
```

Awesome! With this change, you should see all eight tests passing. Let's take a look at the next section to understand how it works.

How it works...

This recipe contained a lot of knowledge from the previous recipes of this chapter. However, the key highlight is the fact that the `CounterService` class that is injected in the component can only be directly mocked using the `jest.spyOn` method. This method removes the need to mock every single service separately. In the first (replaced) test, we spy on the `getFromStorage` method of the `component.counterService` property. This makes it a Jest spy that we can provide to the `expect` block and that can run the `expect(component.counterService.getFromStorage).toHaveBeenCalled();` statement. In the second test, notice that we not only spy on the `getFromStorage` method but also return a mock value, `12`, so that when the `component.ngOnInit` method is called, the value returned from the `CounterService` is this mocked value, `12`. Hence, the `counter` property is set to 12 and the test passes.

Finally, in the last test, we spy on the `saveToStorage` method. Then, every time we click `Increment`, `Decrement`, or `Reset`, we expect the spied function to be called with the correct value. *Easy peasy!*

See also

- Testing Angular Services: `https://angular.io/guide/testing-services`

Mocking child components and directives using the ng-mocks package

Unit tests mostly revolve around testing components, directives, pipes, or services in isolation. However, what if your component depends completely on another component or directive to work properly, especially in a non-standalone application/component? In such cases, you usually provide a mock implementation for the component, but that is a lot of work. However, with the `ng-mocks` package, it is super-easy. In this recipe, we'll learn an advanced example of how to use `ng-mocks` for a parent component that depends on a child component to work properly.

Getting ready

The app that we are going to work with now resides in `start/apps/chapter10/ng-test-ng-mocks` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve ng-test-ng-mocks
   ```

   This should open the app in a new browser tab, and you should see the following:

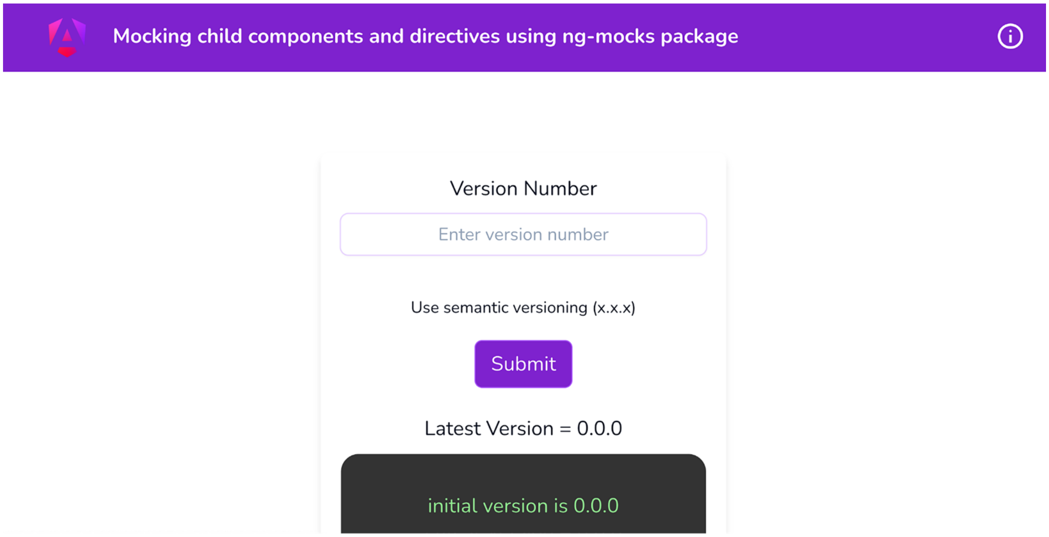
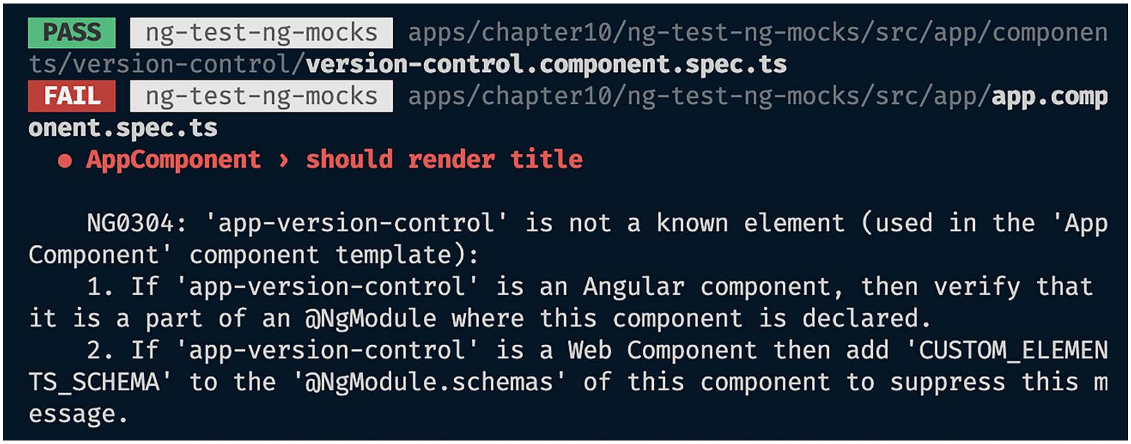

Mocking child components and directives using ng-mocks package
Version Number
Enter version number
Use semantic versioning (x.x.x)
Submit
Latest Version = 0.0.0
initial version is 0.0.0

Figure 10.9: ng-test-ng-mocks app running on http://localhost:4200

Now that we have the app running locally, in the next section, let's go through the steps of the recipe.

How to do it...

If you run the command `npm run test ng-test-ng-mocks` from the workspace root, you'll see that not all of our tests pass. Additionally, there are a bunch of errors on the console, as follows:



```
 PASS   ng-test-ng-mocks  apps/chapter10/ng-test-ng-mocks/src/app/componen
ts/version-control/version-control.component.spec.ts
 FAIL   ng-test-ng-mocks  apps/chapter10/ng-test-ng-mocks/src/app/app.comp
onent.spec.ts
  ● AppComponent › should render title

    NG0304: 'app-version-control' is not a known element (used in the 'App
Component' component template):
    1. If 'app-version-control' is an Angular component, then verify that
it is a part of an @NgModule where this component is declared.
    2. If 'app-version-control' is a Web Component then add 'CUSTOM_ELEMEN
TS_SCHEMA' to the '@NgModule.schemas' of this component to suppress this m
essage.
```

Figure 10.10: Error during unit tests mentioning app-version-control component

Note that we are using an app with `NgModule`, i.e., *not a standalone app*. This is to demonstrate how to work in such situations. Let's go through the recipe to mock the respective components using the `ng-mocks` package:

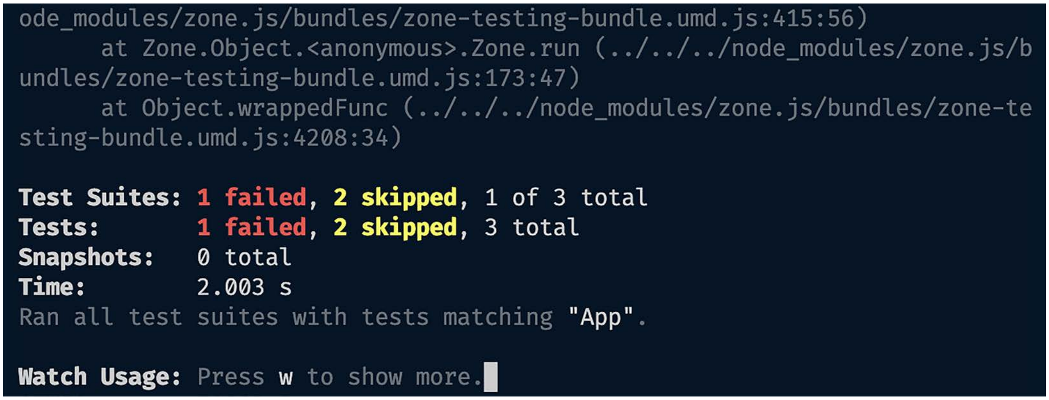
1. First, let's install the `ng-mocks` package within our project. To do this, run the following command from your project root in the Terminal:

   ```
   cd start && npm install ng-mocks --save
   ```

2. Now, we'll try to fix the tests for the `App` component. To only run specific tests based on a string regex, we can use the `-t` parameter with the test command. Run the following command, and after it runs, press the `-t` key to provide the regex `"App"`:

   ```
   npm run test ng-test-ng-mocks -t "App"
   ```

Now, you can see that we only run the tests for `AppComponent`, and they fail as follows:



```
ode_modules/zone.js/bundles/zone-testing-bundle.umd.js:415:56)
      at Zone.Object.<anonymous>.Zone.run (../../../node_modules/zone.js/b
undles/zone-testing-bundle.umd.js:173:47)
      at Object.wrappedFunc (../../../node_modules/zone.js/bundles/zone-te
sting-bundle.umd.js:4208:34)

Test Suites: 1 failed, 2 skipped, 1 of 3 total
Tests:       1 failed, 2 skipped, 3 total
Snapshots:   0 total
Time:        2.003 s
Ran all test suites with tests matching "App".

Watch Usage: Press w to show more.
```

Figure 10.11: Only running specific tests

3. To fix the error shown in *Figure 10.10*, we'll import `VersionControlComponent` into the `TestBed` definition inside the `app.component.spec.ts` file. This is so that our test environment also knows the missing `<app-version-control>` component. To do this, modify the file as follows:

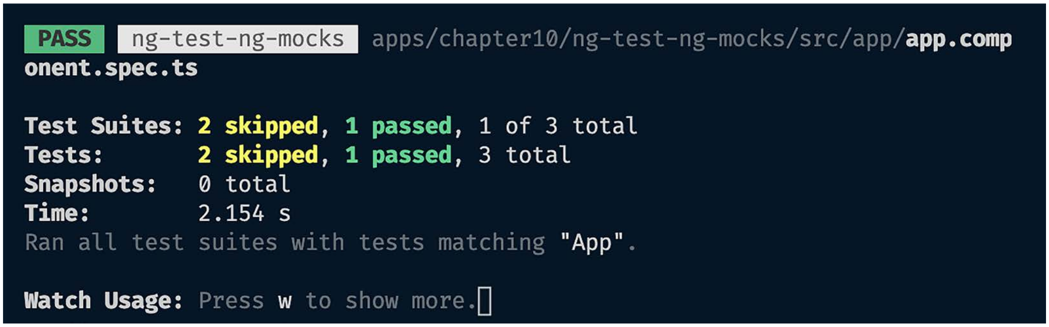
```
...
import { VersionControlComponent } from './components/version-control/
version-control.component';
...
describe('AppComponent', () => {
  beforeEach(waitForAsync(() => {
    TestBed.configureTestingModule({
      imports: [RouterTestingModule],
      declarations: [AppComponent,
        VersionControlComponent],
    }).compileComponents();
  }));
  ...
});
```

If you rerun the tests for `AppComponent`, you'll see some fresher and newer errors. Progress! But since we have the `Version Control` component dependent on another component, the tests don't pass. We'll discuss why this is in more detail in the *How it works...* section. However, to fix this, let's follow the next steps.

4. Instead of providing the `VersionControlComponent` directly, we need to mock it, since we don't really care about it for the tests for `AppComponent`. One way would be to use `CUSTOM_ELEMENT_SCHEMA` in the `TestBed`'s configuration to tell the test bed that we don't really care about the `VersionControlComponent` inside the tests for `AppComponent`. However, we're going to use a `MockComponent` from `ng-mocks` to mock the component, achieving the same outcome. To do this, update the `app.component.spec.ts` file as follows:

```
...
import { MockComponent } from 'ng-mocks';
...
describe('AppComponent', () => {
  beforeEach(waitForAsync(() => {
    TestBed.configureTestingModule({
      declarations: [AppComponent,
        MockComponent(VersionControlComponent)],
      imports: [RouterTestingModule],
    }).compileComponents();
  }));
  ...
});
```

Boom! Problem solved. Run the tests again, just for the `AppComponent`, and you should see them all pass as follows:



```
PASS  ng-test-ng-mocks  apps/chapter10/ng-test-ng-mocks/src/app/app.comp
onent.spec.ts

Test Suites: 2 skipped, 1 passed, 1 of 3 total
Tests:       2 skipped, 1 passed, 3 total
Snapshots:   0 total
Time:        2.154 s
Ran all test suites with tests matching "App".

Watch Usage: Press w to show more.
```

Figure 10.12: Passing all of the tests for App component

5. Press the *a* key in the terminal to run all tests again.
6. Now, let's talk about the tests for the `Version Control` component. This depends on the `VC Logs` component. Let's mock the `VCLogsComponent` class like a pro this time, using the `MockBuilder` and `MockRender` methods, so we can get rid of the errors during the tests. After the update, the `version-control.component.spec.ts` file should appear as follows:

```
import { ComponentFixture, TestBed, waitForAsync } from '@angular/core/
testing';
import { VersionControlComponent } from './version-control.component';
import { FormsModule } from '@angular/forms';
import { MockBuilder, MockRender, MockedComponentFixture } from 'ng-
mocks';
import { AppModule } from '../../app.module';
import { VcLogsComponent } from '../vc-logs/vc-logs.component';

describe('VersionControlComponent', () => {
  let component: VersionControlComponent;
  let fixture: MockedComponentFixture
    <VersionControlComponent>;

  beforeEach(() => {
    return MockBuilder(
      VersionControlComponent,
      AppModule
    );
  });
  beforeEach(() => {
    fixture = MockRender(VersionControlComponent);
    component = fixture.point.componentInstance;
  });

  it('should create', () => {...});
});
```

If you run `npm run test` now, you should see all of the tests passing. Let's write another test in the next step.

7. `VersionControlComponent` uses `VCLogsComponent` as a child component in the template. Additionally, it provides the `vName` property as `@Input()` to `VCLogsComponent`, via the `[vName]` attribute. We can check whether the input's value is set correctly. To do so, update the `version-control.component.spec.ts` file, as follows:

```
import { VersionControlComponent } from './version-control.component';
import { MockBuilder, MockRender, MockedComponentFixture, ngMocks } from
'ng-mocks';
import { AppModule } from '../../app.module';
import { VcLogsComponent } from '../vc-logs/vc-logs.component';
describe('VersionControlComponent', () => {
  ...
  it('should create', () => {...});

  it('should set vName input value in VCLogsComponent',
    () => {
    const vcLogsComponent = ngMocks.find<VcLogsComponent>(
      'app-vc-logs'
    ).componentInstance;
    component.versionName = '2.2.2';
    fixture.detectChanges();
    expect(vcLogsComponent.vName).toBe('2.2.2');
  });
});
```

8. We are now going to modify the `vc-logs.component.spec.ts` file to make sure that when the `vName` changes inside the `VCLogsComponent`, we have a new log created inside the `logs` array. To do so, modify the file as follows:

```
...
import { TestBed, waitForAsync } from '@angular/core/testing';
import { VcLogsComponent } from './vc-logs.component';
import { MockRender, MockedComponentFixture } from 'ng-mocks';

describe('VcLogsComponent', () => {
  ...
  beforeEach(waitForAsync(() => {...}));

  beforeEach(() => {
    fixture = MockRender(VcLogsComponent, {
      vName: '0.0.0'
    });
    component = fixture.point.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should add a log after vName change', () => {
    fixture.detectChanges();
    fixture.componentInstance.vName = "2.2.3";
    fixture.detectChanges();
    expect(component.logs).toHaveLength(2);
  });
});
```

Boom! We have implemented some interesting tests by using the `ng-mocks` package. I absolutely love it every time I use it (mostly with components that are part of an `NgModule`, as `standalone` components don't have this issue). Now that we've finished the recipe, in the next section, let's take a look at how it all works.

How it works...

There are a couple of interesting things that we have covered in this recipe. First, to avoid any errors on the console complaining about unknown components, we use the `MockComponent` method from the `ng-mocks` package, declaring the components we're dependent on as mocks. Notice that when a component is mocked, it loses its functionality, as all the methods become `spy` functions. That is absolutely the simplest thing we can achieve with the `ng-mocks` package. However, we will move on to an advanced situation, which I will admit is sort of an unconventional approach, that is, testing the `@Input` properties and `@Output` emitters of the child components in the parent component in order to test an entire flow. This is what we do for the tests of `VersionControlComponent` by checking the value of an `@Input` property `vName`. Again, this is mostly useful for non-standalone components.

Notice that we removed the usage of the `@angular/core/testing` package completely from the `version-control.component.spec.ts` file. This is because we're no longer using `TestBed` to create the test environment. Instead, we use the `MockBuilder` method from the `ng-mocks` package to build the test environment for our `VersionControlComponent`. The `MockBuilder` method has several overloads, the one we used to take the target component as the first argument, and the `NgModule` this component is a part of as the second argument. The module and everything inside it are mocked. This makes it so much easier to test the component in isolation. You also have methods like `exclude` that you can chain, like `MockBuilder(MyComponent, MyModule).exclude(OtherComponent)`, to not mock specific components. However, we didn't need to do that in this recipe. We also wrote an interesting test for the `VersionControlComponent`, that is, getting a child component (`VCLogsComponent`) using the `ngMocks.find` method and checking if the child component's `@Input()` is set appropriately when the respective property in the parent changes. I call this interesting because now we're getting away from the entire notion of "testing components in isolation," as we are now checking a property of another component than the component we're actually testing. In general, you would want to mock the child components or any dependency that a component has. With that said, the approach in your team might be different. If you have a strong case for not mocking the child component and then checking its properties as shown in the recipe, now you know how. Please refer to the next section for further resources to read.

IMPORTANT NOTE

Notice that we don't use a spy on the `VersionControlComponent.addNewReleaseLog` method. This is because if we do so, that function will become a Jest spy function. Therefore, it'll lose its internal functionality. In return, it'll never add the new log to the `releaseLogs` array, and none of our tests will pass. You can try it out for fun.

See also

- The `ng-mocks` package's `mock` method: `https://ng-mocks.sudo.eu/api/MockBuilder#mock`
- The `ng-mocks` official documentation: `https://ng-mocks.sudo.eu`

Writing even easier tests with Angular CDK component harnesses

When writing tests for components, there might be scenarios where you'd want to interact with the DOM elements. Now, this can already be achieved by using the `fixture.debugElement.query` method to find the element, using a selector and then triggering events on it. However, that means maintaining the DOM queries for different platforms, knowing the identifiers of all the selectors, and then exposing all of that in the tests. And this is even worse if we're talking about an Angular library. It certainly isn't necessary for each developer who interacts with my library to know all the element selectors in order to write tests. Only the author of the library should know that much to respect encapsulation. Luckily, we have the component harnesses from the Angular CDK team, which were released with Angular 9 along with the IVY compiler. And they've led by example, by providing component harnesses for the Angular material components as well. In this recipe, you'll learn how to create your own component harnesses.

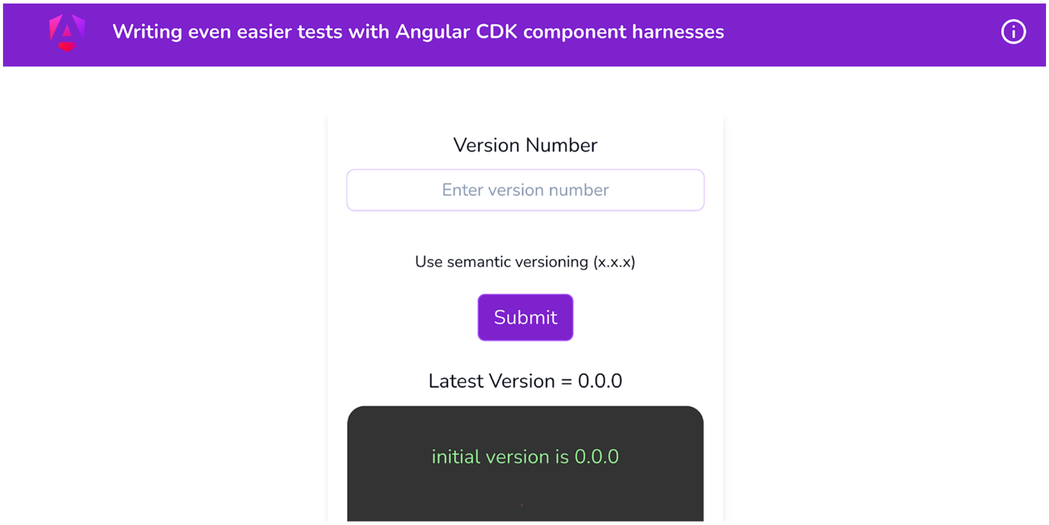
Getting ready

The app that we are going to work with now resides in `start/apps/chapter10/ng-test-cdk-harness` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve ng-test-cdk-harness
   ```

   This should open the app in a new browser tab, and you should see the following:



Writing even easier tests with Angular CDK component harnesses

Version Number

Enter version number

Use semantic versioning (x.x.x)

Latest Version = 0.0.0

initial version is 0.0.0

Figure 10.13: The ng-test-cdk-harness app running on http://localhost:4200

Now that you have the app running, let's move on to the next section to follow the recipe.

How to do it...

We have our favorite Angular version control app that allows us to create release logs. Also, we have tests written already, including tests that interact with the DOM element to validate a few use cases. Let's follow the recipe to use component harnesses instead, discovering how easy it becomes to use in the actual tests:

1. First, run the following command to run the tests:

   ```
   npm run test ng-test-cdk-harness
   ```

2. Now, we'll create a `component harness` for the `VersionControlComponent`. Let's create a new file inside the `version-control` folder, and name it `version-control.component.harness.ts`. Then, add the following code inside it:

   ```
   import { ComponentHarness } from '@angular/cdk/testing';
   export class VersionControlComponentHarness extends
     ComponentHarness {
     static hostSelector = 'app-release-form';
     protected getSubmitButton =
       this.locatorFor('button[type=submit]');
     protected getAppVersionInput =
       this.locatorFor(`#versionNumber`);
     protected getVersionErrorEl =
       this.locatorFor('small.error');
   }
   ```

3. Now, we need to set up the harness environment for our tests for `VersionControlComponent`. For this, we'll use `TestbedHarnessEnvironment` from the Angular CDK. Update the `version-control.component.spec.ts` file as follows:

   ```
         import { TestbedHarnessEnvironment } from '@angular/cdk/testing/
   testbed';
         ...
   ```

4. Now, let's write some methods in our `VersionControlComponentHarness` class to get the relevant information. We'll use these methods in the later steps. Update the `version-control.component.harness.ts` file as follows:

   ```
   ...
   export class ReleaseFormComponentHarness extends ComponentHarness {
     ...
     async clickSubmit() {
       const submitBtn = await this.getSubmitButton();
       return await submitBtn.click();
   ```

```
  }
  async setNewAppVersion(version: string) {
    const versionInput = await this.getAppVersionInput();
    return await versionInput.sendKeys(version);
  }
  async isVersionErrorShown() {
    const expected = 'Version number does not match the
      pattern (x.x.x)';
    const versionErrorEl = await this.getVersionErrorEl();
    const versionErrorText = await versionErrorEl.text();
    const isErrorShown = versionErrorText.trim() ===
      expected;
    if (!isErrorShown) {
      console.log({
        actual: versionErrorText.trim(),
        expected,
      });
    }
    return isErrorShown;
  }
```

5. Next, we'll work on our first test, named `should show error on wrong version number input`, with the component harness. Update the `version-control.component.spec.ts` file as follows:

```
...
import { VersionControlComponentHarness } from './version-control.
component.harness';
describe('VersionControlComponent', () => {
  ...
  it('should show error on wrong version number input',
    async () => {
    const vcHarness = await TestbedHarnessEnvironment
      .harnessForFixture(
      fixture,
      VersionControlComponentHarness
    );
    await vcHarness.setNewAppVersion('abcd');
    await vcHarness.clickSubmit();
    fixture.detectChanges();
    const isErrorShown = await vcHarness
      .isVersionErrorShown();
      expect(isErrorShown).toBe(true);
    });
    ...
  });
```

Now, if you run `npm run test`, you should see all of the tests passing, which means our first test with the component harness works. Woo-hoo!

6. For the final test, we also need a component harness for `VCLogsComponent`. Let's quickly create it. Add a new file inside the `vc-logs` folder, named `vc-logs.component.harness.ts`, and add the following code:

```
import { ComponentHarness } from '@angular/cdk/testing';
export class VCLogsComponentHarness extends ComponentHarness {
  static hostSelector = 'app-vc-logs';
  protected getLogsList = this.locatorForAll('.logs__item');

  async getLogsLength() {
    const logsElements = await this.getLogsList();
    return logsElements.length;
  }

  protected async getLogTextAt(index: number) {
    const logsElements = await this.getLogsList();
    return (await logsElements[index].text()).trim();
  }

  async getFirstLogText() {
    return await this.getLogTextAt(0);
  }

  async getSecondLogText() {
    return await this.getLogTextAt(1);
  }
}
```

7. Finally, let's modify our final tests in the `version-control.component.spec.ts` file as follows:

```
...
import { VersionControlComponentHarness } from './version-control.
component.harness';
import { VCLogsComponentHarness } from '../vc-logs/vc-logs.component.
harness';
```

```
describe('VersionControlComponent', () => {
  ...
  it('should show the new log in the list after adding
    submitting a new log', async () => {
    const vcHarness = await TestbedHarnessEnvironment
      .harnessForFixture(
      fixture,
      VersionControlComponentHarness
    );
    const harnessLoader = TestbedHarnessEnvironment
      .loader(fixture);
    const vcLogsHarness = await harnessLoader.getHarness(
      VCLogsComponentHarness
    );
    const VERSION = '2.3.6';
    await vcHarness.setNewAppVersion(VERSION);
    await vcHarness.clickSubmit();
    fixture.detectChanges();
    const logsLength = await vcLogsHarness.getLogsLength();
    expect(logsLength).toBe(2);
    const firstLogText = await vcLogsHarness
      .getFirstLogText();
    expect(firstLogText).toBe(`initial version is 0.0.0`);
    const secondLogText = await vcLogsHarness
      .getSecondLogText();
    expect(secondLogText).toBe(`version changed to
      2.3.6`);
  });
});
```

Voilà! That's some amazing testing right there using the Angular CDK component harnesses. If you run the tests now, you should see them all passing. Now that you've finished the recipe, please refer to the next section to learn how this works.

How it works...

All right! That was a cool recipe, and I enjoyed writing it too. The key factor of this recipe is the `@angular/cdk/testing` package. If you have worked with `e2e` tests using Protractor before, this is a similar concept to the **Pages** or **Page Objects** (**POs**) in Protractor. The first thing we do is to create a component harness for both the `VCLogsComponent` and the `VersionControlComponent`.

Notice that we import the `ComponentHarness` class from `@angular/cdk/testing` for both component harnesses. Then, we extend our custom classes, called `VersionControlComponentHarness` and `VCLogsComponentHarness`, from the `ComponentHarness` class. Essentially, this is the correct way to author component harnesses. Did you notice the static property called `hostSelector`? We need this property for every component harness class that we create, and the value is always the selector of the target element/component. This ensures that when we load this harness into the test environment, the environment can find the host element in the DOM – for which we're creating the component harness. In our component harness class, we use the `this.locatorFor` method to find elements within the host component. The `locateFor` method takes a single argument as the **CSS selector** of the element to be found, returning an `AsyncFactoryFn`. This means the returned value is a function that we can use later to get the required elements.

In the `VersionControlComponentHarness` class, we find the submit button and the version number input using the *protected* methods `getSubmitButton` and `getAppVersionInput`, respectively. All of these are of the `AsyncFactoryFn` type, as mentioned earlier. We have these methods set as *protected* because we don't want the developers who write the unit tests to access or care about the information of the DOM elements. This makes it much easier for everyone to write tests without worrying about the internal implementation of accessing the DOM.

One important thing to mention here is that when we call the `locatorFor` method or the `locatorForAll` method, we get back a `Promise` with the `TestElement` item or a `Promise` with a list of `TestElement` items, respectively. Each `TestElement` item has a bunch of handy methods, such as `click`, `sendKeys`, `focus`, `blur`, `getProperty`, `text`, and more. These methods are what we're interested in, since we use them behind the scenes to interact with the DOM elements.

Now, let's talk about configuring the test environment. In the `version-control.component.spec.ts` file, we set up the environment to use component harnesses for both `VCLogsComponent` and `VersionControlComponent`. The `TestbedHarnessEnvironment` element is the key element here. We use the `TestbedHarnessEnvironment.harnessForFixture` method first. This is because we are writing tests for the `VersionControlComponent`, and we also want the harness of the same component as root. Therefore, the `harnessForFixture` method makes sense here. This method takes the `fixture` as the first argument and the `harness` class as the second. Then, we also use the `loader` method of the `TestbedHarnessEnvironment` class to get an instance of the `HarnessLoader`. This loader can then get the harness of the `VCLogsComponent`, i.e., get the instance of the `VCLogsComponentHarness` class. Note that the fixture is what we get in the test environment using the `TestBed.createComponent(VersionControlComponent)` statement.

Notice that in the tests, we use the `harnessLoader.getHarness` method by providing the harness class as an argument. This enables the test environment to find the DOM element associated with the `hostSelector` property of the harness class. Additionally, we get back the instance of the component harness that we can use further in the test.

See also

- Finding components in the DOM with component harnesses: `https://material.angular.io/cdk/test-harnesses/overview#finding-elements-in-the-components-dom`

- API for component harness authors: `https://material.angular.io/cdk/test-harnesses/overview#api-for-component-harness-authors`

Unit-testing responses from HTTP calls

If you're building an Angular application, it is very likely that you'll work with HTTP calls inside the app at some point. For instance, you could be fetching data from a third-party API or just making APIs to your own server. In either case, it becomes slightly difficult to test applications that have HTTP calls in action. In this recipe, we're going to learn how to create unit-tests with HTTP calls.

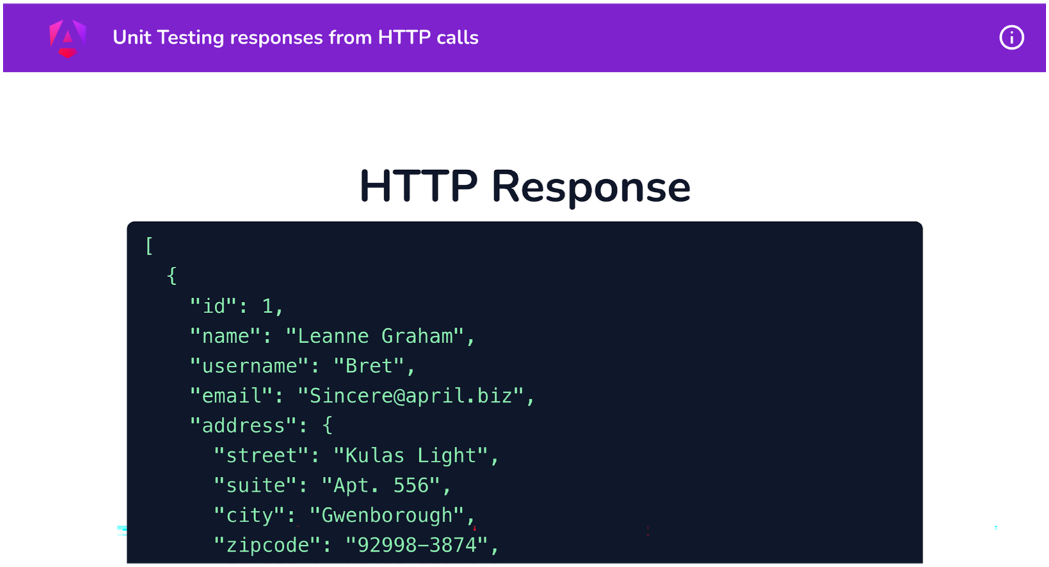
Getting ready

The app that we are going to work with now resides in `start/apps/chapter10/ng-test-http-resp` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-test-http-resp
```

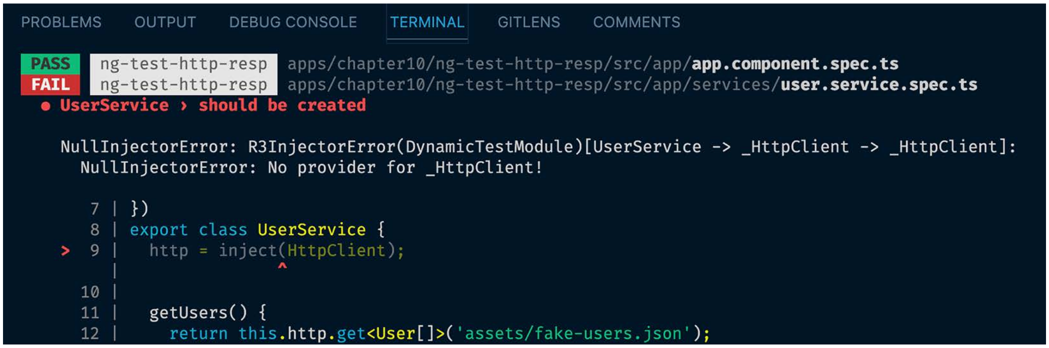
This should open the app in a new browser tab, and you should see the following:



```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
```

Figure 10.14: The ng-test-http-resp app running on http://localhost:4200

Let's also run the tests. Right now, the test for the `UserService`, which fetches the `HTTP` response, doesn't pass. You should see an output similar to *Figure 10.15*:



```
PASS  ng-test-http-resp  apps/chapter10/ng-test-http-resp/src/app/app.component.spec.ts
FAIL  ng-test-http-resp  apps/chapter10/ng-test-http-resp/src/app/services/user.service.spec.ts
 ● UserService › should be created

   NullInjectorError: R3InjectorError(DynamicTestModule)[UserService -> _HttpClient -> _HttpClient]:
     NullInjectorError: No provider for _HttpClient!

       7 | })
       8 | export class UserService {
   >   9 |   http = inject(HttpClient);
         |                 ^
      10 |
      11 |   getUsers() {
      12 |     return this.http.get<User[]>('assets/fake-users.json');
```

Figure 10.15: Test failing for the UserService

Now that we have the app and tests running locally, in the next section, let's take a look at the steps of the recipe.

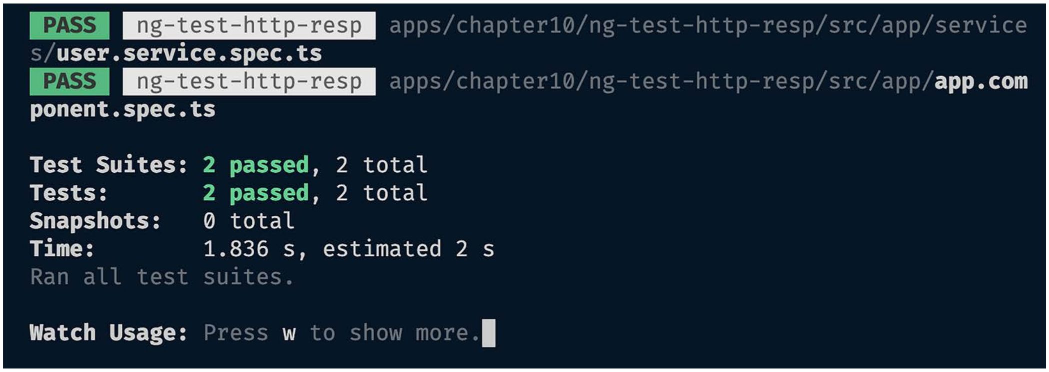
How to do it...

We'll write one single test that checks if the service's `getUser` method returns the correct data. If you open the `UserService`, you'll notice that the service fetches the users using an HTTP call and then modifies the data to add the `fullAddress` property. Our test should expect the same to happen. Let's get started:

1. First and foremost, we'll fix the failing test. We'll add the `HttpClientModule` class to the test in the `imports` array. Update the `user.service.spec.ts` as follows:

```
...
import { HttpClientModule } from '@angular/common/http';

describe('UserService', () => {
  ...
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientModule],
    });
    service = TestBed.inject(UserService);
  });
  ...
});
```

Now, as soon as you run the `npm run test` command, you'll see that the tests pass now as follows:



```
 PASS   ng-test-http-resp  apps/chapter10/ng-test-http-resp/src/app/service
s/user.service.spec.ts
 PASS   ng-test-http-resp  apps/chapter10/ng-test-http-resp/src/app/app.com
ponent.spec.ts

Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.836 s, estimated 2 s
Ran all test suites.

Watch Usage: Press w to show more.
```

Figure 10.16: Tests passing for the UserService as well

2. Now, we'll import the `HttpClientTestingModule` in the `user.service.spec.ts` file to be able to use it to intercept HTTP calls. Modify the file as follows:

```
...
import { HttpClientTestingModule,
} from '@angular/common/http/testing';
...
describe('UserService', () => {
  let service: UserService;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientModule, HttpClientTestingModule],
    });
    service = TestBed.inject(UserService);
  });
});
```

3. Let's create a controller using the `HttpTestingController` class from the same `@angular/common/http/testing` package. We'll use this controller later to expect and intercept the HTTP calls. Update the `user.service.spec.ts` file as follows:

```
...
import {
  HttpClientTestingModule,
  HttpTestingController,
} from '@angular/common/http/testing';

describe('UserService', () => {
  let service: UserService;
  let httpTestingController: HttpTestingController;
  beforeEach(() => {
    TestBed.configureTestingModule({...});
    service = TestBed.inject(UserService);
    httpTestingController = TestBed.inject(
      HttpTestingController);
  });
});
```

4. Now, we'll write a test (finally). We'll intercept the HTTP call to `/assets/users.json` using the `httpTestingController`, and then we will compare the result with our expected data. Update the `user.service.ts` file by adding a test, as follows:

```
...
import { User } from '../user.interface';
describe('UserService', () => {
  ...
  it('should return expected user data (HttpClient called
    once)', () => {
    const mockUsers: User[] = [];
    const req = httpTestingController
      .expectOne('assets/users.json');
    expect(req.request.method).toEqual('GET');

    req.flush(mockUsers); // Respond with mocked data
  });
});
```

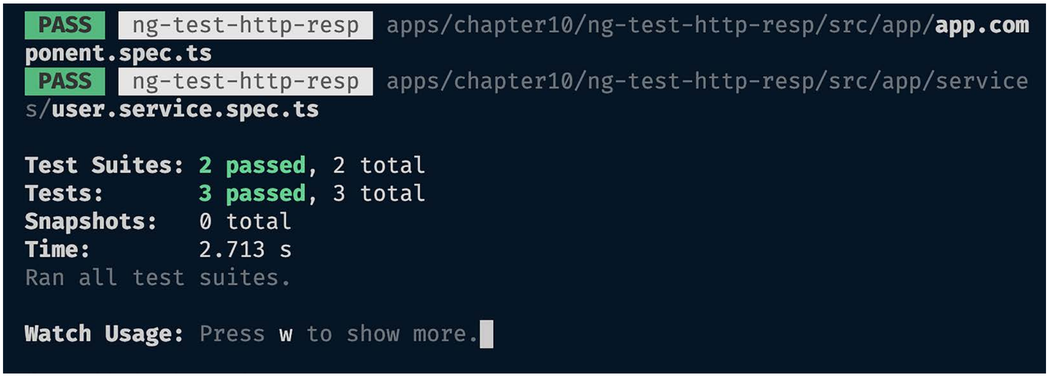
5. We'll add an `afterEach` hook in the tests to verify that there are no pending requests after our test. Let's add it to the `user.service.spec.ts` file as follows:

```
...
describe('UserService', () => {
  ...
  afterEach(() => {
    // After every test, assert that there are no more
    pending requests.
    httpTestingController.verify();
  });
  ...
});
```

6. Now, let's populate the mock data (the `mockUsers` array) with two users. Since we're using `req.flush(mockUsers)`, the HTTP call will be intercepted, and this data (`mockUsers` array) will be returned from it as a response. Replace the test `it('should return expected user data (HttpClient called once)'` in the file `user.service.spec.ts` with the snippet from `https://ng-cookbook.com/s/ng-test-http-mock-users`.
7. Finally, update the test further to make the http call, using the `getUsers` method, and see if we can expect the transformed data. Update the `user.service.spec.ts` file, as follows:

```
describe('UserService', () => {
  ...
  it('should return expected user data (HttpClient called
    once)', (done) => {
    const mockUsers: User[] = [...];
    service.getUsers().subscribe({
      next: (data) => {
        expect(data).toEqual([{
          ...mockUsers[0],
          fullAddress: 'sample street 1, 123 ABC, Dream
            city, 4567'
        }, {
          ...mockUsers[1],
          fullAddress: 'sample street 2, 123 ABC, Dream
            city, 890'
        }]);
        done();
      },
      error: (err) => {
        console.log('Error: ', err);
      },
    });
    ...
  });
});
```

If you run the tests, you should see them all pass as follows:



```
 PASS   ng-test-http-resp  apps/chapter10/ng-test-http-resp/src/app/app.com
ponent.spec.ts
 PASS   ng-test-http-resp  apps/chapter10/ng-test-http-resp/src/app/service
s/user.service.spec.ts

Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.713 s
Ran all test suites.

Watch Usage: Press w to show more.
```

Figure 10.17: All of the tests pass

Great! You now know how to work with `HttpTestingController` to test services that have methods, using HTTP calls. Although there's still a lot to learn about testing HTTP calls and Observables in Angular, the purpose of this recipe was to keep everything simple and sweet.

Now that you have finished the recipe, please refer to the next section to understand how it works.

How it works...

The hero of this recipe is the `HttpClientTestingModule`, which allows us to use an instance of `HttpTestingController`. The `HttpTestingController` makes it easy to intercept the HTTP calls in our code and allows us to return a specific response. This makes it easier for us to not have our Angular services tested with real HTTP calls, which can be expensive. Another benefit of this approach to mocking the response is always having the same response object to work with. In our case, our `getUsers` method not only made an HTTP call but also transformed data, using the `map` operator from `RxJs`. Therefore, we used an array of mocked users and expected the result from the `getUsers` method to have the transformed data, i.e., including the `fullAddress` property containing the right data. You can see in the test that we use the `HttpTestingController.expectOne` method, telling our test that there should be only one request when we run the test. We expect the method of that request to be `GET`. Then, we expect that the returned value from the `getUsers` method has an array of users containing the `fullAddress` property.

See also

- Angular testing component scenarios: `https://docs.angular.lat/guide/testing-components-scenarios`
- `HttpTestingController` docs: `https://angular.io/api/common/http/testing/HttpTestingController`

Unit-testing Angular pipes

In my opinion, pipes are the easiest thing to test in an Angular application. Why? Well, this is because they're (supposed to be) pure functions that return the same result based on the same set of inputs. In this recipe, we'll write some tests for a simple pipe in an Angular application.

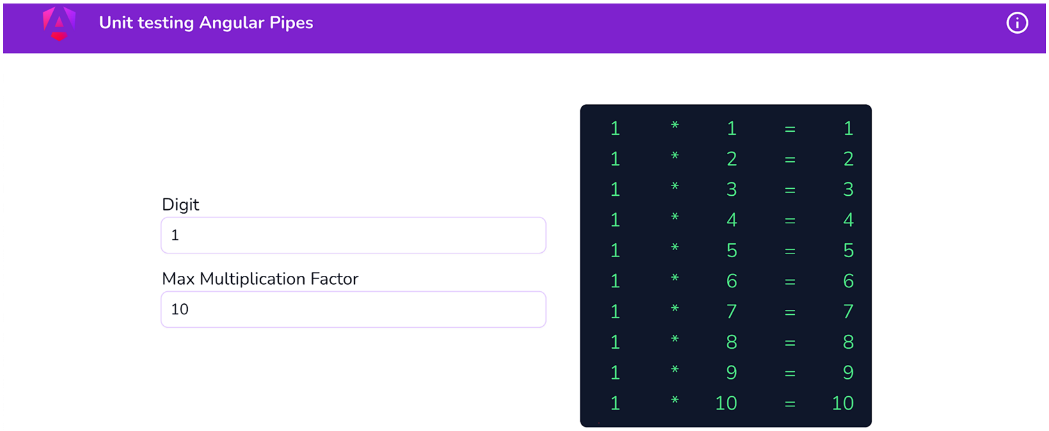
Getting ready

The app that we are going to work with now resides in `start/apps/chapter10/ng-test-pipes` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-test-pipes
```

This should open the app in a new browser tab, and you should see the following:



Digit
1

Max Multiplication Factor
10

```
1  *  1  =  1
1  *  2  =  2
1  *  3  =  3
1  *  4  =  4
1  *  5  =  5
1  *  6  =  6
1  *  7  =  7
1  *  8  =  8
1  *  9  =  9
1  *  10 =  10
```

Figure 10.18: The ng-test-pipes app running on http://localhost:4200

Now that we have the app running locally, in the next section, let's go through the steps of the recipe.

How to do it...

Here, we have a simple recipe that takes two inputs – the digit and the max factor value. Based on these inputs, we show a multiplication table. We already have the `MultTablePipe` that is working fine according to our business logic. We'll now write some unit tests to validate our inputs and expected outputs, as follows:

1. Let's write our first custom test for `MultTablePipe`. We'll make sure it returns an empty array when we have an invalid value for the `digit` input. Replace the whole `mult-table.pipe.spec.ts` file with the following code:

```
import { MultTablePipe } from './mult-table.pipe';
describe('MultTablePipe', () => {
  let pipe: MultTablePipe;
  beforeEach(() => {
    pipe = new MultTablePipe();
  })
  it('should return an empty array if the value of digit
    is not valid', () => {
    const digit = 0;
    const limit = 10;
    const outputArray = pipe.transform(null, digit, limit);
    expect(outputArray).toEqual([]);
  });
});
```

2. Let's write another test to validate the `limit` input so that we also return an empty array if it is invalid:

```
...
describe('MultTablePipe', () => {
  ...
  it('should return an empty array if the value of limit
    is not valid', () => {
    const digit = 10;
    const limit = 0;
    const outputArray = pipe.transform(null, digit,
      limit);
    expect(outputArray).toEqual([]);
  });
});
```

3. Now, we'll write a test to validate the output of the pipe's `transform` method when both the `digit` and `limit` inputs are valid. In this scenario, we should get back the array containing the multiplication table. Write another test as follows:

```
      describe('MultTablePipe', () => {
  ...
  it('should return the correct multiplication table
    when both digit and limit inputs are valid', () => {
    const digit = 10;
    const limit = 2;
    const expectedArray = [{ digit: 10, factor: 1, result:
      10 },
    { digit: 10, factor: 2, result: 20 },];
    const outputArray = pipe.transform(null, digit, limit);
    expect(outputArray).toEqual(expectedArray);
  });
});
```

4. Right now, within the app, we have the possibility to provide decimal digits for the `limit` input. For instance, we can write `2.5` as the max factor in the input. To handle this, we use a `Math.floor` in `MultTablePipe` to round it down to the lower number. Let's write a test to make sure this works:

```
...
describe('MultTablePipe', () => {
  ...
  it('should round of the limit if it is provided
    in decimals', () => {
    const digit = 10;
    const limit = 3.5;
    const expectedArray = [
      {digit: 10, factor: 1, result: 10},
      {digit: 10, factor: 2, result: 20},
      {digit: 10, factor: 3, result: 30}
    ]; // rounded off to 3 factors instead of 3.5
    const outputArray = pipe.transform(null, digit, limit);
    expect(outputArray).toEqual(expectedArray);
  });
});
```

Easy peasy! Writing tests for Angular pipes is so straightforward that I love it. We could call this the power of pure functions. Now that you've finished the recipe, please refer to the next section for more informative links.

See also

- Testing Angular pipes official documentation: `https://angular.io/guide/testing-pipes`
- Test Angular Pipes With Services: `https://levelup.gitconnected.com/test-angular-pipes-with-services-4cf77e34e576`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

11

E2E Tests in Angular with Cypress

An app having a couple of **End-to-End** (**E2E**) tests surely promises more reliability than an app having no tests at all, and in today's world, with emerging businesses and complex applications, it becomes essential at some point to have E2E tests written to capture the entire flow of an application. Cypress is one of the best tools out there today when it comes to E2E tests for web applications. In this chapter, you'll learn how to test your E2E flows in an Angular app with Cypress. Here are the recipes we're going to cover in this chapter:

- Writing your first Cypress test
- Validating if a DOM element is visible on the view
- Testing form inputs and submission
- Waiting for **XMLHttpRequests** (**XHRs**) to finish
- Using Cypress bundled packages
- Using Cypress fixtures to provide mock data

Technical requirements

For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter11`.

Writing your first Cypress test

If you have been writing E2E tests already, you might have been doing this using Protractor. Working with Cypress is a completely different experience, though. In this recipe, you'll set up Cypress with an existing Angular application and write your first E2E test with Cypress.

Getting ready

The app that we are going to work with resides in `start/apps/chapter11/ng-cypress-starter` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-cypress-starter
```

This should open the app in a new browser tab, and you should see the following:

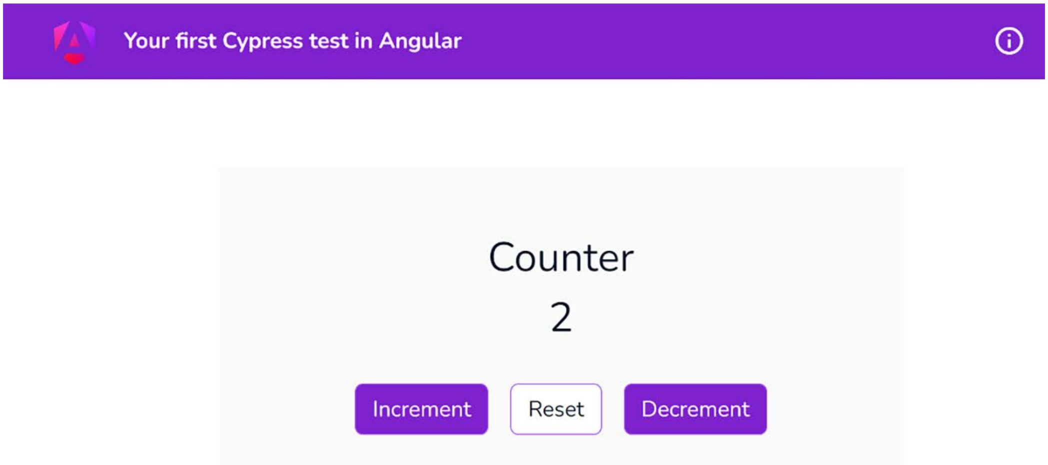


Figure 11.1: ng-cypress-starter app running on localhost:4200

How to do it...

The app we're working with is a simple counter application. It has minimum and maximum values and some buttons that can increment, decrement, and reset the counter's value. We'll start by configuring Cypress for our application and will then move toward writing the test:

1. Since we're using an NX workspace, the setup here looks different from a regular Angular application. Open a new terminal window/tab and make sure you're inside the workspace's root folder. Once inside, run the following command to install **Cypress** in our project:

```
cd start && npm install --save-dev @nx/cypress
```

2. Now, run the following command from the workspace's root folder to create a `cypress` project for the app as follows:

```
cd start && nx g @nx/cypress:cypress-project ng-cypress-starter-e2e
--project=ng-cypress-starter --directory apps/chapter11/ng-cypress-
starter-e2e
```

When (or if) asked, choose the `Vite` bundler for Cypress during the installation since Vite is a faster bundler and provides a faster development server as well. And select the "Ap provided" so we have the folder `ng-cy-starter-e2e` created in `start/apps/chapter11` folder. You'll see that a new folder named `ng-cypress-starter-e2e` will be created inside the `start/apps` folder.

3. Let us run a script to rename our app from `chapter11-ng-cypress-starter-e2e` to `ng-cypress-starter-e2e`. This will make it easier for us to run the `e2e` tests for this recipe and the next. Use the following command from the workspace's root folder:

```
node scripts/rename-app.js chapter11 ng-cypress-starter-e2e start
```

4. Now you can run the following command to start the Cypress tests from the workspace's root folder (outside of the `start` folder):

```
npm run e2e ng-cypress-starter
```

You should be able to use a browser to start running the tests. I will be using Chrome as the browser for `e2e` tests in this book.

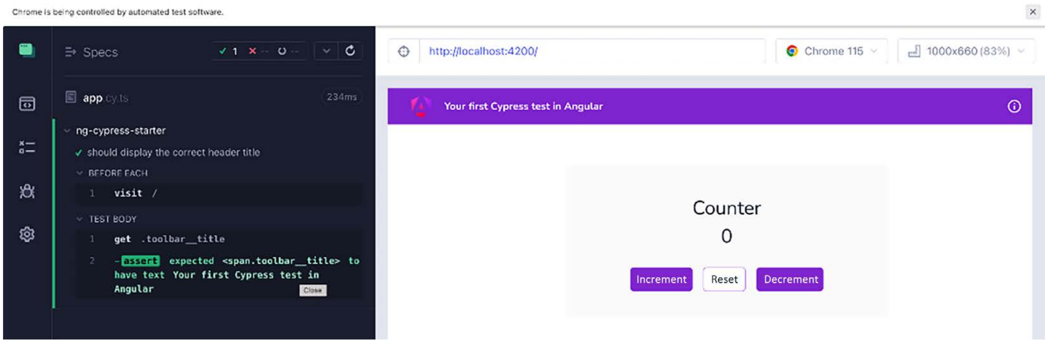
5. Click the `app.cy.ts` in the `cypress-chrome` window (the browser window opened as a result of *step 4*) to run the default-created tests. We will be modifying this file in the recipe to write our own tests. Once you run the tests, you'll see them failing. But do not worry about it since we have not written our own tests.
6. Let us create our first test now. We will just check whether the title in the header of our app says `Your first Cypress test in Angular`. Let us replace the entire content of the `src/e2e/support/app.po.ts` file by creating a **PO** (**Page Object**) in the file as follows:

```
export const getHeaderTitle = () =>
  cy.get('.toolbar__title');
```

7. We will now import the `getHeaderTitle` in the `src/e2e/app.cy.ts` file and replace the first test as follows:

```
import { getHeaderTitle } from '../support/app.po';

describe('ng-cypress-starter', () => {
  beforeEach(() => cy.visit('/'));

  it('should display the correct header title', () => {
    getHeaderTitle().should('contain.text','Your first
      Cypress test in Angular');
  });
});
```

8. If you look again at the Cypress window, you should see the test passing as follows:



Chrome is being controlled by automated test software.
Specs
✓ 1 ✕ -- ○ --
http://localhost:4200/
Chrome 115
1000x660 (83%)
app.cy.ts
234ms
Your first Cypress test in Angular
Counter
0
Increment
Reset
Decrement
Close

```
ng-cypress-starter
✓ should display the correct header title
BEFORE EACH
1  visit /
TEST BODY
1  get .toolbar__title
2  -assert expected <span.toolbar__title> to
   have text Your first Cypress test in
   Angular
```

Figure 11.2: Our first Cypress test passes

Easy, right? Now that you know how to configure Cypress for an Angular app (especially in NX), see the next section to understand how it works.

How it works...

Cypress can be integrated with absolutely any framework and web development project. One interesting fact is that Cypress uses Mocha as the test runner behind the scenes. The tooling for Cypress watches for code changes so that you do not have to recompile the tests repeatedly. Cypress also adds a shell around the application being tested to capture logs and access DOM elements during the tests, and some functionality for debugging tests.

At the very top of our `app.cy.ts` file, we use the `describe` method, which defines the test suite, defining the context of the tests about to be written inside. Then, we use a `beforeEach` method to specify what should happen before each test is executed. Since each test starts with no data, we first must make sure that Cypress navigates to our application's URL: `http://localhost:4200`. The reason we just specify `cy.visit('/')` and it still works is that NX automagically configures it with the `@nx/cypress` package. If you add Cypress to a standard Angular app (not in an NX workspace), you will have to specify the `baseUrl` in the Cypress config file (`cypress.config.ts`) as follows:

```
import { defineConfig } from 'cypress'
export default defineConfig({
  e2e: {
    baseUrl: 'http://localhost:4200',
  },
})
```

However, NX does this for us for the recipes of this book. Therefore, we just have to provide relative URLs in our tests.

For each of our tests, we use the `it` method to specify the title for them. You will notice that in the `app.cy.ts` file, we are importing the `getHeaderTitle` method from the `support/app.po` file. As mentioned earlier in the recipe, **PO** means **page object**. This is a widespread practice to use these objects to contain functions that return **Document Object Module** (**DOM**) elements. This keeps our tests clear of code that interacts with the DOM to retrieve elements, and we have reusable functions for our tests. In the `app.po` file, you can see that we use the `cy.get` method to retrieve a single element that has the class `toolbar__title` applied to it. All the Angular applications for the recipes in this book have a header and a title that shows what the recipe is about. Notice that in the `app.cy.ts` file, we use the `getHeaderTitle` method to get the target element from our HTML page. And then we use the `should()` method to compare the text of the title with the expected value `Your first Cypress test in Angular`. Notice that we are using ‘`contain.text`' instead of `'have.text'` because there can be whitespace in the target element. The following are some other examples of the `should` method with some different statements:

- `should('be.visible')`
- `should('be.empty')`
- `should('be.visible')`
- `should('have.class''my-class')`
- `should('have.id''newUserId')`
- `should('be.visible')`
- `should('have.focus')`

Now that you understand how the recipe works, see the next section for some useful links.

See also

- `cy.get` documentation: `https://docs.cypress.io/api/commands/get`
- Cypress documentation – W*riting Your First Test*: `https://docs.cypress.io/guides/getting-started/writing-your-first-test.html`

Validating if a DOM element is visible on the view

In most web applications, there is at least one element/view that is shown based on a certain condition. Otherwise, it is hidden. When it comes to making sure the end users see the right content in the right situations, it becomes necessary to test them well. In this recipe, you are going to learn how to check if an element is visible on the DOM.

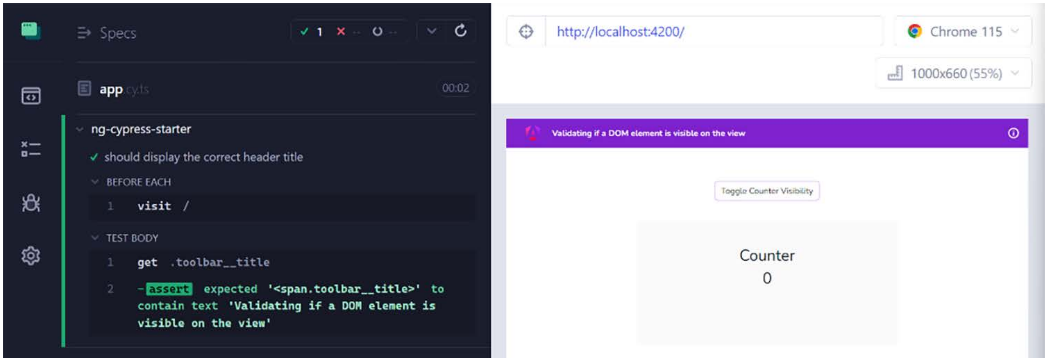
Getting ready

The app that we are going to work with resides in `start/apps/chapter11/ng-cy-element-visibility` inside the cloned repository. However, the `e2e` tests are in the folder `start/apps/chapter11/ng-cy-element-visibility-e2e`. In this recipe, we are going to modify files in both folders. Let’s run the e2e tests first by following these steps:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to run the e2e tests for the project:

```
npm run e2e ng-cy-element-visibility
```

This should open the Cypress window. Select Chrome for testing and click the `app.cy.ts` file to run the tests and you should see the following:



ng-cypress-starter

✓ should display the correct header title

BEFORE EACH

```
1 visit /
```

TEST BODY

```
1 get .toolbar__title
2 -assert expected '<span.toolbar__title>' to contain text 'Validating if a DOM element is visible on the view'
```

Figure 11.3: Cypress tests running for the ng-cy-element-visibility app

Now that we have the app and the Cypress tests running locally, let us see the steps of the recipe in the next section.

How to do it...

We have the same old counter app from the previous recipe. However, some things have changed. We now have a button at the top that toggles the visibility of the counter component (`CounterComponent`). Also, we must hover over the counter card to see the **Increment**, **Decrement**, and **Reset** action buttons.

1. Let's create the page objects necessary for our tests. We'll create functions to return the toggle button and the counter card from the page object `app.po.ts` file. Update the `src/support/app.po.ts` file in the `start/apps/chapter11/ng-cy-element-visibility-e2e` folder as follows:

```
export const getHeaderTitle = () =>
  cy.get('.toolbar__title');

export const getToggleCounterButton = () =>
  cy.get('[data-test-id="toggleCounterBtn"]');

export const getCounterCard = () => cy.get(
  '[data-test-id="counterCard"]');

export const getCounterActions = () => getCounterCard()
  .find('button');
```

2. Let's add the relevant test IDs now to the HTML. We'll add the `test-id` attributes for both the toggle-counter button and the counter element. Modify the file `ng-cy-element-visibility/src/app/app.component.html` as follows:

```
...
<main class="content" role="main">
  <div class="...">
    <button data-test-id="toggleCounterBtn"
       (click)="toggleCounterVisibility()">Toggle Counter
        Visibility</button>
    <app-counter data-test-id="counterCard"
       *ngIf="visibility ===
       visibilityOptions.Visible"></app-counter>
  </div>
</main>
```

3. Now, we will write a test to ensure that our counter card is shown and hidden when we click the toggle-counter button. To do this, update the `ng-cy-element-visibility-e2e/src/e2e/app.cy.ts` file as follows:

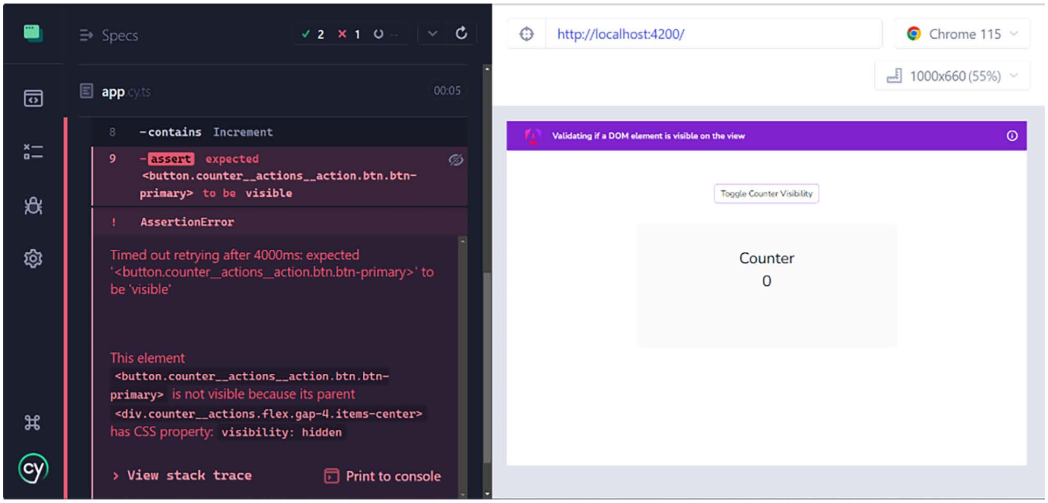
```
import { getCounterCard, getHeaderTitle, getToggleCounterButton } from
'../support/app.po';

describe('ng-cypress-starter', () => {
  beforeEach(() => cy.visit('/'));

  it('should display the correct header title', () => ...});

  it('should toggle visibility of counter card when the
    toggle button is clicked', () => {
    getCounterCard().should('exist');
    getToggleCounterButton().click();
    getCounterCard().should('not.exist');
    getToggleCounterButton().click();
    getCounterCard().should('exist');
  });
});
```

4. Now, we'll write another test to check if our action buttons (`Increment`, `Decrement`, and `Reset`) show up when we hover over the `Counter` component. Update the `app.cy.ts` file again, as follows:

```
import { getCounterActions, getCounterCard, getHeaderTitle,
getToggleCounterButton } from '../support/app.po';
```

```
describe('ng-cypress-starter', () => {
  ...
  it('should show the action buttons when the counter card is hovered',
    () => {
    getCounterCard().trigger('mouseover');
    getCounterActions().should('have.length', 3);
    getCounterActions().contains('Increment')
      .should('be.visible');
    getCounterActions().contains('Decrement')
      .should('be.visible');
    getCounterActions().contains('Reset')
      .should('be.visible');
  });
});
```

If you look at the Cypress window now, you should see the test failing, as follows:


Specs
✓ 2 ✕ 1
app.cy.ts
00:05
http://localhost:4200/
Chrome 115
1000x660 (55%)
Validating if a DOM element is visible on the view
Toggle Counter Visibility
Counter
0

```
8   -contains  Increment
9   -assert  expected <button.counter__actions__action.btn.btn-primary> to be visible
!   AssertionError
```

Timed out retrying after 4000ms: expected '<button.counter__actions__action.btn.btn-primary>' to be 'visible'

This element `<button.counter__actions__action.btn.btn-primary>` is not visible because its parent `<div.counter__actions.flex.gap-4.items-center>` has CSS property: `visibility: hidden`

> View stack trace
Print to console

Figure 11.4: Unable to get action buttons on hovering

The reason for the test's failure is that Cypress does not currently provide a **CSS** hover effect. To work around this, we will install a package in the next step.

5. Stop running the e2e tests and then install the `cypress-real-events` package, as follows from the workspace's root folder:

```
cd start && npm install --save-dev cypress-real-events
```

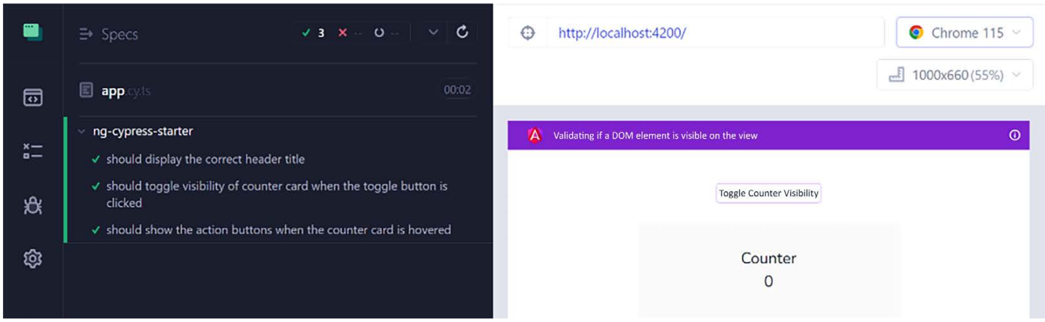
6. Now, open the `src/support/e2e.ts` file in the `ng-cy-element-visibility-e2e` project and update it, as follows:

```
...
// Import commands.js using ES2015 syntax:
/// <reference types="cypress-real-events" />
import './commands';
import 'cypress-real-events/support';
...
```

7. Now, update the `app.cy.ts` file to use the `realHover` method from the package on the `counter card` element, as follows:

```
...
describe('ng-cypress-starter', () => {
  ...

  it('should show the action buttons when the counter card is hovered', () =>
    {
    getCounterCard().realHover();
    ...
  })
});
```

8. Now, run the `npm run e2e ng-cy-element-visibility` command again from the workspace root (if it is not running already). You should see all the tests passing, as shown in *Figure 11.5*:



ng-cypress-starter

✓ should display the correct header title

✓ should toggle visibility of counter card when the toggle button is clicked

✓ should show the action buttons when the counter card is hovered

Validating if a DOM element is visible on the view

Counter

0

Figure 11.5: All tests passing

Awesome! You've just learned how to check the visibility of DOM elements in different scenarios. These are, of course, not the only options available to identify and interact with DOM elements. You can refer to the Cypress documentation for more possibilities. Now that you have finished the recipe, see the next section to understand how it works.

How it works...

We started the recipe by building the POs. It is a good idea to do so to have the functions ready for when we write the tests. And if a function doesn't exist to get a particular page object, we can create it on the go. Notice that we check `getCounterCard` with a `should('exist')` statement. If you don't already know, we could have used `should('be.visible')` as well, and that would work too. But when we want to make sure that it is not visible, we can't use the `should('not.be.visible')` statement. And now you'll be thinking, "What??" Exactly! Since `'visible'` in Cypress is built so that the element exists in the DOM and is visible too, it can't accommodate cases where an element doesn't exist in the DOM if we use `'be.visible'` or `'not.be.visible'`. And since we use the `*ngIf` directive to show or hide our `Counter component`, it ends up either *existing* in the DOM or not. Hence, using `should('exist')` and `should('not.exis')` is the appropriate choice here.

For the next test, we wanted to see if hovering (or doing a mouse over) on the counter card displays the action buttons. For this, we could use the `trigger` method on the counter card with the `mouseover` event. However, that won't work. Why? Because all the hover workarounds in Cypress eventually lead to triggering the JavaScript events and doesn't affect the CSS pseudo selectors, and since we have our action buttons (with the `'.counter__actions__action'` selector) shown on the `:hover` (CSS) of the element with the `'.counter'` selector, our tests fail because, in the tests, our action buttons are not actually shown. To tackle the issue, we use the `cypress-real-events` package, which has the `realHover` method that affects the pseudo selectors and eventually shows our action buttons.

See also

- Cypress official documentation on the visibility of items: `https://docs.cypress.io/guides/core-concepts/interacting-with-elements.html#Visibility`
- `cypress-real-events` project repository: `https://github.com/dmtrKovalenko/cypress-real-events`

Testing form inputs and submission

If you're building a web app, there's a high chance that you're going to have at least one form in it, and when it comes to forms, we need to make sure that we have the right **User Experience** (**UX**) and the right business logic in place. What better way to make sure everything works as expected than writing E2E tests for them? In this recipe, we're going to test a form using Cypress and validate if the correct errors are being shown in appropriate situations.

Getting ready

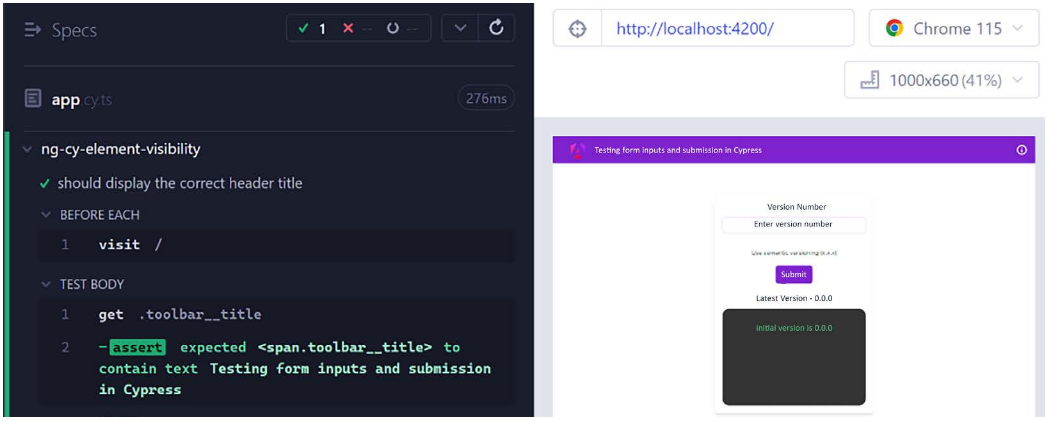
The app that we are going to work with resides in `start/apps/chapter11/ng-cy-forms` inside the cloned repository. However, the e2e tests are in the folder `start/apps/chapter11/ng-cy-forms-e2e`. In this recipe, we're going to modify files in both folders. Let's run the e2e tests first by following these steps:

1. Open the code repository in your code editor.

2. Open the terminal, navigate to the code repository directory, and run the following command to run the e2e tests for the project:

```
npm run e2e ng-cy-forms
```

This should open the Cypress window. Select Chrome for testing and click the `app.cy.ts` file to run the tests and you should see the following:



Specs
✓ 1 ✕ -- ○ --
app.cy.ts
276ms
http://localhost:4200/
Chrome 115
1000x660 (41%)
Testing form inputs and submission in Cypress
Version Number
Enter version number
Submit
Latest Version - 0.0.0
initial version is 0.0.0

```
ng-cy-element-visibility
  ✓ should display the correct header title
    BEFORE EACH
      1   visit  /
    TEST BODY
      1   get  .toolbar__title
      2   -assert  expected <span.toolbar__title> to
          contain text Testing form inputs and submission
          in Cypress
```

Figure 11.6: Cypress tests running for the app ng-cy-forms

Now that we have the Cypress tests running, let's see the steps of the recipe in the next section.

How to do it...

We have to make sure that we see a **new log** when the form is successfully submitted with a new version. We also need to make sure that we see relevant errors if the version input is empty, or if the value of the version input is an invalid value. Let's get started:

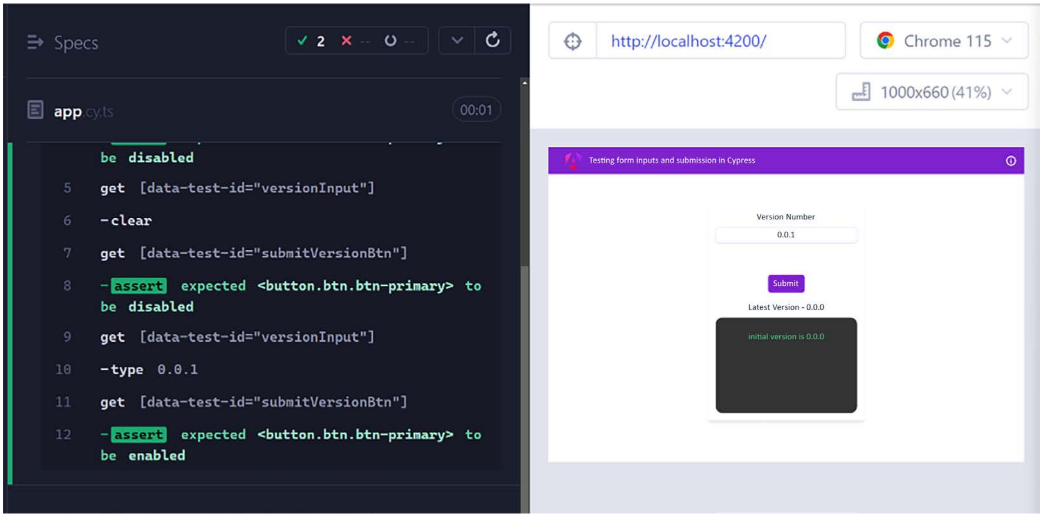
1. Let's create the page objects required for our tests. We already have the `data-test-id` attribute on the items we want to use in the tests. So, we can reference them in the page object file. Update the `start/apps/chapter11/ng-cy-forms-e2e/src/support/app.po.ts` file as follows:

```
export const getHeaderTitle = () => cy.get('.toolbar__title');
export const getVersionInput = () => cy.get('[data-test-
id="versionInput"]');
export const getRequiredError = () => cy.get('[data-test-
id="versionReqErr"]');
export const getMismatchError = () => cy.get('[data-test-
id="versionMismatchErr"]');
export const getSubmitButton = () => cy.get('[data-test-
id="submitVersionBtn"]');
export const getLogsListItems = () => cy.get('[data-test-id="logsList"]
.logs__item');
export const getLatestVersion = () => cy.get('[data-test-
id="latestVersion"]');
```

2. We will first validate that our form cannot be submitted unless we have a valid version. To do that, let's make sure that the **Submit** button is disabled when the input gets cleared after typing, or when there's an invalid value. Open the `src/e2e/app.cy.ts` file in the e2e project and add a test, as follows:

```
import { getHeaderTitle, getSubmitButton, getVersionInput } from '../
support/app.po';

describe('ng-cy-forms', () => {
  beforeEach(() => cy.visit('/'));

  it('should display the correct header title', () => {...});

  it('should have the submit button disabled on invalid input', () => {
    getVersionInput().type('invalid input');
    getSubmitButton().should('be.disabled');
    getVersionInput().clear();
    getSubmitButton().should('be.disabled');
    getVersionInput().type('0.0.1');
    getSubmitButton().should('be.enabled');
  });
});
```

If you look at the Cypress window and exapand the tests, you should see the tests passing, as follows:


Specs
✓ 2 ✕ -- ○ --
app.cy.ts
00:01
http://localhost:4200/
Chrome 115
1000x660 (41%)
Testing form inputs and submission in Cypress
Version Number
0.0.1
Submit
Latest Version - 0.0.0
initial version is 0.0.0

```
      be disabled
 5    get [data-test-id="versionInput"]
 6    -clear
 7    get [data-test-id="submitVersionBtn"]
 8    -assert expected <button.btn.btn-primary> to
      be disabled
 9    get [data-test-id="versionInput"]
10    -type 0.0.1
11    get [data-test-id="submitVersionBtn"]
12    -assert expected <button.btn.btn-primary> to
      be enabled
```

Figure 11.7: Checking if the Submit button is disabled when there is invalid input

3. Let's add another test that validates that we see a new version log when submitting a valid version. Add another test in the `app.cy.ts` file, as follows:

```
...
import { getHeaderTitle, getLatestVersion, getLogsListItems,
getSubmitButton, getVersionInput } from '../support/app.po';

describe('ng-cy-forms', () => {
  ...
  it('should add a new version log on valid version submission', () => {
    getLogsListItems().should('have.length', 1);
    getVersionInput().type('0.0.1');
    getSubmitButton().click();
    getLogsListItems().should('have.length', 2);
    getLogsListItems().eq(1).then(el => {
      expect(
        el.text().trim()
      ).to.eq('version changed to 0.0.1')
    });
  });
});
```

4. We'll add another test now to make sure that we can see the latest version above the version logs. Let's modify the `app.cy.ts` file, as follows:

```
...
describe('ng-cy-forms', () => {
  ...

  it('should display the latest version', () => {
    getLatestVersion().should('have.text', 'Latest Version = 0.0.0');
    getVersionInput().type('0.0.1');
    getSubmitButton().click();
    getLatestVersion().should('have.text', 'Latest Version = 0.0.1');
  });
});
```

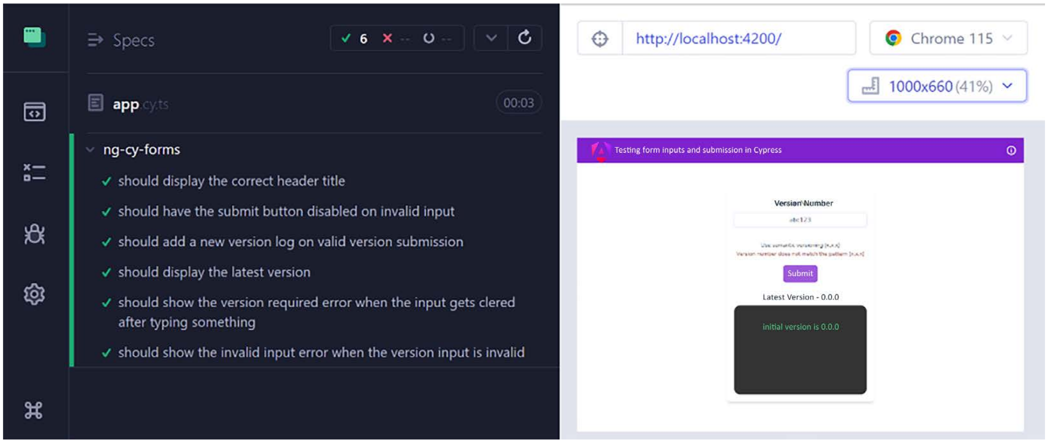
5. We will now add a test that will verify if the user sees the error `'Version number is required'` when the version input is cleared after typing something, i.e., before the value is submitted. Add the test in the `app.cy.ts` file, as follows:

```
import { getHeaderTitle, getLatestVersion, getLogsListItems,
getRequiredError, getSubmitButton, getVersionInput } from '../support/
app.po';

describe('ng-cy-forms', () => {
  ...
  it('should show the version required error when the input
  gets clered after typing something', () => {
    getVersionInput().type('0.0.1');
    getVersionInput().clear();
    getRequiredError().should('exist');
    getRequiredError().should('be.visible');
    getRequiredError().then(el => {
      expect(
        el.text().trim()
      ).to.eq('Version number is required')
    });
  });
});
```

6. Finally, let us write a test to make sure we show an error message on invalid inputs. Add another test in the `app.cy.ts` file, as follows:

```
...
import { getHeaderTitle, getLatestVersion, getLogsListItems,
getMismatchError, getRequiredError, getSubmitButton, getVersionInput }
from '../support/app.po';

describe('ng-cy-forms', () => {
  ...
  it('should show the invalid input error when the
  version input is invalid', () => {
    getVersionInput().type('abc123');
    getMismatchError().should('exist');
    getMismatchError().should('be.visible');
    getMismatchError().then(el => {
      expect(
        el.text().trim()
      ).to.eq('Version number does not match the pattern (x.x.x)')
      });
    });
  });
```

If you look at the **Tests** window now, you should see all the tests passing, as follows:



Specs
✓ 6 ✕ -- ↻ --
http://localhost:4200/
Chrome 115
1000x660 (41%)
app.cy.ts
00:03
Testing form inputs and submission in Cypress
Version Number
abc123
Submit
Latest Version - 0.0.0
initial version is 0.0.0

ng-cy-forms
- ✓ should display the correct header title
- ✓ should have the submit button disabled on invalid input
- ✓ should add a new version log on valid version submission
- ✓ should display the latest version
- ✓ should show the version required error when the input gets clered after typing something
- ✓ should show the invalid input error when the version input is invalid

Figure 11.8: All tests passing for the app

Awesome! You now know how to use Cypress to test forms with some interesting use cases and assertions. See the next section to understand how it all works.

How it works...

We start by implementing some page objects in our `app.po.ts` file since we can reuse those methods to get the elements easily. Since our app's logic has a rule that the **Submit** button should be disabled until we have a valid version in the version input, we use the `'be.disabled'` assertion on the **Submit** button, as follows:

```
getSubmitButton().should('be.disabled');
```

We then use the `getVersionInput().type('...')` function to type the desired values in the version input and check if the button is disabled either when we have an invalid value for the version input, or no value entered at all.

Then we check if there is a new log added to the logs list when a valid version is submitted. The important chunk of code for this test is as follows:

```
getLogsListItems().eq(1).then(el => {
  expect(
     el.text().trim()
  ).to.eq('version changed to 0.0.1')
});
```

Notice that we get the list of logs, i.e., the log items. Then we use `eq(1)` to get the second element from the list. Then we use the `then` method to get the `jQuery<HTMLElement>` so we can use the `trim` method on the text content of the element. This is because when writing HTML templates for Angular applications, we might end up writing the content in the HTML tags in a formatted way that ends up including whitespaces in the text content. So, it is a clever idea to trim the text before using the `expect` function against the expected value. Alternatively, you can use the `.should('contain.text', 'EXPECTED_TEXT')` assertion instead of the `.should('have.text', 'EXPECTED_TEXT')` assertion.

For the cases where we want to check that the appropriate errors are being displayed, we ensure the following:

- The error element exists on the DOM
- The error element is visible to the user
- The error element has the appropriate error message

Notice that we use the `then` method to get the error elements to trim the text content before our test assertions, like what we did for the log item's text content validation.

See also

- Cypress recipes – Form interactions: `https://github.com/cypress-io/cypress-example-recipes/tree/master/examples/testing-dom__form-interactions`
- Cypress recipes – Login form: `https://github.com/cypress-io/cypress-example-recipes/tree/master/examples/logging-in__html-web-forms`

Waiting for XHRs to finish

Testing **User Interface** (**UI**) transitions is the essence of E2E testing. While it is important to test the predicted outcome of an action right away, there might be cases where the outcome has a dependency. For instance, if a user fills out the **Login** form, we can't show the success toast until we have a successful response from the backend server, hence we can't test whether the success toast is shown right away. In this recipe, you're going to learn how to wait for a specific XHR call to be completed before performing an assertion.

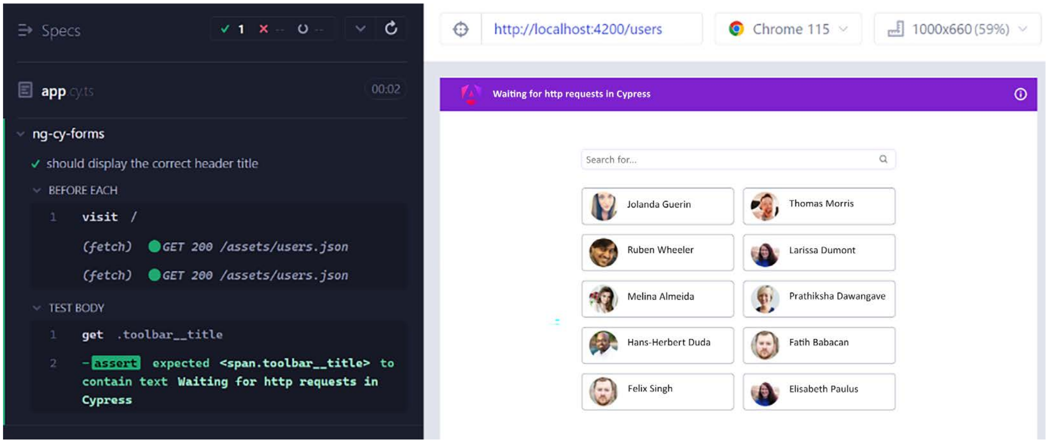
Getting ready

The app that we are going to work with resides in `start/apps/chapter11/ng-cy-http-requests` inside the cloned repository. However, the e2e tests are in the folder `start/apps/chapter11/ng-cy-http-requests-e2e`. In this recipe, we're going to modify files for the e2e project only. Let's run the e2e tests by following these steps:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to run the e2e tests for the project:

```
npm run e2e ng-cy-http-requests
```

This should open the Cypress window. Select Chrome for testing and click the `app.cy.ts` file to run the tests, and you should see the following:



```
ng-cy-forms
✓ should display the correct header title
BEFORE EACH
visit /
(fetch) GET 200 /assets/users.json
(fetch) GET 200 /assets/users.json
TEST BODY
get .toolbar__title
-assert expected <span.toolbar__title> to contain text Waiting for http requests in Cypress
```

Specs
✓ 1 ✕ -- ○ --
app.cy.ts
00:02
http://localhost:4200/users
Chrome 115
1000x660 (59%)
Waiting for http requests in Cypress
Search for...
Jolanda Guerin
Thomas Morris
Ruben Wheeler
Larissa Dumont
Melina Almeida
Prathiksha Dawangave
Hans-Herbert Duda
Fatih Babacan
Felix Singh
Elisabeth Paulus

Figure 11.9: Cypress tests running for the ng-cy-http-requests app

Now that we have the Cypress tests running, let's see the steps of the recipe in the next section.

How to do it...

We are going to start with some tests that will work just fine. However, if there's a delay in the response for the HTTP calls, they will fail. That is because Cypress has a timeout of 4,000 **milliseconds** (**ms**) (4 seconds), during which it tries the assertion again and again until the assertion passes. What if our XHR takes more than 4,000 ms? Let's try it out in the recipe:

1. We will write our test first. We'll make sure that we get 10 users from the HTTP call's response. But before that, we will create the desired page objects for this recipe in the `users.po.ts` file, as follows:

```
export const getUsersCards = () => {
  return cy.get('app-users ul li');
}

export const getSearchInput = () => {
  return cy.get('[data-test-id="searchUsersInput"]');
}
```

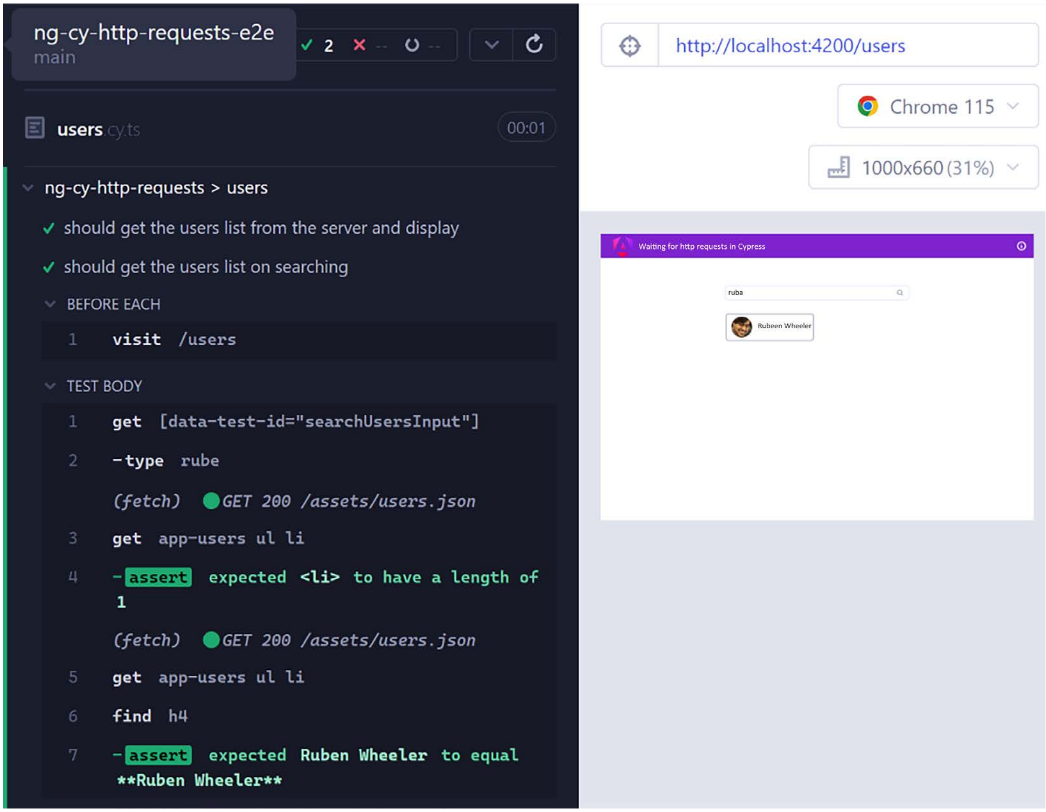
2. Update the users.cy.ts file to add the following test:

```
import { getUsersCards } from '../support/users.po';

describe('ng-cy-http-requests > users', () => {
  beforeEach(() => cy.visit('/users'));

  it('should get the users list from the server and display', () => {
    getUsersCards().should('have.length', 10);
  });
});
```

3. We will write another test to check if we get the searched users based on the search input's value. Add another test in the `users.cy.ts` file as follows:

```
import { getSearchInput, getUsersCards } from '../support/users.po';

describe('ng-cy-http-requests > users', () => {
  ...
  it('should get the users list on searching', () => {
    getSearchInput().type('rube');
    getUsersCards().should('have.length', 1);
    getUsersCards().find('h4').should(
      el => {
        expect(
          el.text().trim()
        ).to.eq(
          'Ruben Wheeler'
        )
      }
    );
  });
});
```

You should see both tests passing, as shown in *Figure 11.10*. However, this is not the best way of writing UI tests as they should be *independent* of the data received from actual API servers. In practice, we usually mock the API calls, which you will learn about in the *Using Cypress fixtures to provide mock data* recipe later in this chapter.


ng-cy-http-requests-e2e
main
✓ 2 ✕ -- ○ --
http://localhost:4200/users
Chrome 115
1000x660 (31%)
Waiting for http requests in Cypress
ruba
Rubeen Wheeler

users.cy.ts 00:01

ng-cy-http-requests > users

✓ should get the users list from the server and display

✓ should get the users list on searching

BEFORE EACH

```
1  visit  /users
```

TEST BODY

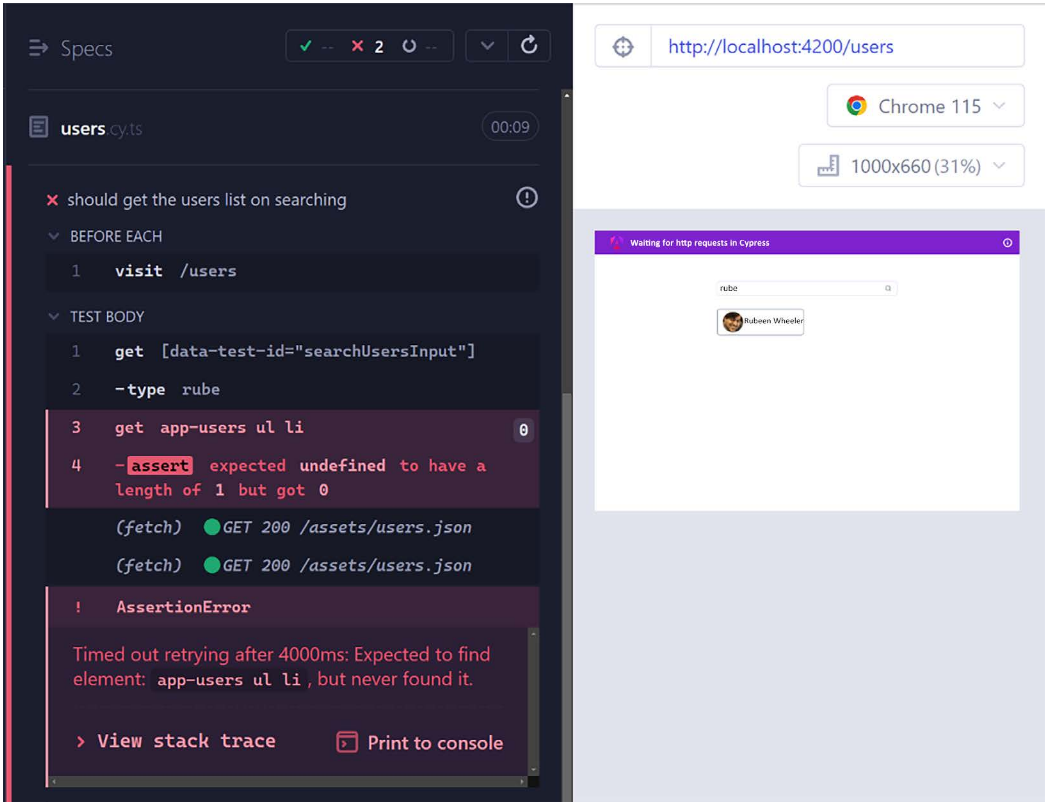
```
1  get  [data-test-id="searchUsersInput"]
2  -type  rube
   (fetch)  GET 200 /assets/users.json
3  get  app-users ul li
4  -assert  expected <li> to have a length of
   1
   (fetch)  GET 200 /assets/users.json
5  get  app-users ul li
6  find  h4
7  -assert  expected Ruben Wheeler to equal
   **Ruben Wheeler**
```

Figure 11.10: Tests for the user page passing

4. First of all, we need to simulate the scenario where the desired result occurs after 4,000 ms. We'll use the `delay` operator from `rxjs` for this, with a delay of 5,000 ms. Let's apply it in the `user.service.ts` file in the project, as follows:

```
…
import { EMPTY, Observable, delay, map, mergeMap, of } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class UserService {
  http = inject(HttpClient);
  getAll(): Observable<User[]> {
    return of(EMPTY)
      .pipe(
        delay(5000),
        mergeMap(() => {
          return this.http.get<User[]>(
            '/assets/users.json')
        })
      );
  }
  ...}
```

If you now check the Cypress tests, you should see a test failing, as shown in *Figure 11.11*:


Specs
✓ -- ✕ 2 ⟳ --
http://localhost:4200/users
Chrome 115
1000x660 (31%)
Waiting for http requests in Cypress
rube
Rubeen Wheeler

users.cy.ts 00:09

✕ should get the users list on searching

BEFORE EACH

```
1  visit  /users
```

TEST BODY

```
1  get  [data-test-id="searchUsersInput"]
2  -type  rube
3  get  app-users ul li                                0
4  -assert  expected undefined to have a
   length of 1 but got 0
   (fetch)  ● GET 200 /assets/users.json
   (fetch)  ● GET 200 /assets/users.json
!  AssertionError
```

Timed out retrying after 4000ms: Expected to find element: `app-users ul li`, but never found it.

> View stack trace   Print to console

Figure 11.11: Assertion failing for the test for searching a particular user

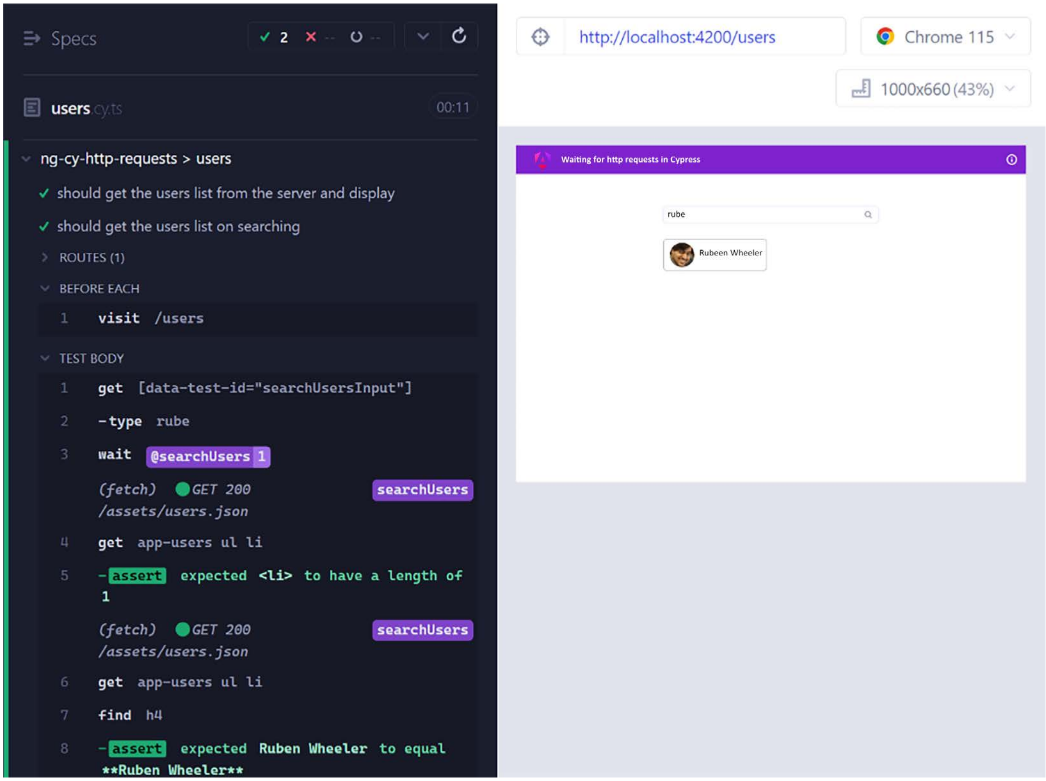
5. We can now try to fix this, so it doesn't matter how long the XHR takes—we'll always wait for it to be completed before making an assertion. Let's intercept the XHR call and create an alias for it so that we can use it later to wait for the XHR call. Update the `users.cy.ts` file, as follows:

```
...
describe('ng-cy-http-requests > users', () => {
  beforeEach(() => {
    cy.intercept('/assets/users.json').as('searchUsers');
    cy.visit('/users');
  });
  ...
});
```

6. Now, let's use the alias to wait for the XHR call to complete before the assertion. Update the `users.cy.ts` file, as follows:

```
...
describe('ng-cy-http-requests > users', () => {
  ...
  it('should get the users list from the server and display', () => {
    cy.wait('@searchUsers', {
      timeout: 10000
    });
    getUsersCards().should('have.length', 10);
  });

  it('should get the users list on searching', () => {
    getSearchInput().type('rube');
    cy.wait('@searchUsers', {
      timeout: 10000
    });
    getUsersCards().should('have.length', 1);
    getUsersCards().find('h4').should(...);
  });
});
```

If you check the Cypress tests now for `users.cy.ts`, you should see all of them pass, as follows:



```
ng-cy-http-requests > users
✓ should get the users list from the server and display
✓ should get the users list on searching
ROUTES (1)
BEFORE EACH
1   visit  /users
TEST BODY
1   get  [data-test-id="searchUsersInput"]
2   -type  rube
3   wait  @searchUsers 1
    (fetch)  GET 200                     searchUsers
    /assets/users.json
4   get  app-users ul li
5   -assert  expected <li> to have a length of
    1
    (fetch)  GET 200                     searchUsers
    /assets/users.json
6   get  app-users ul li
7   find  h4
8   -assert  expected Ruben Wheeler to equal
    **Ruben Wheeler**
```

Figure 11.12: Tests waiting for the XHR call to be completed before the assertion

Great! You now know how to implement E2E tests with Cypress that include waiting for a particular XHR call to finish before an assertion. To understand all the magic behind the recipe, see the next section.

How it works…

In the recipe, we use something called variable aliasing. We first use the `cy.intercept` method so that Cypress can listen to the network call. Note that we use the specific URL that is `/assets/users.json` as the parameter, and then we use the `.as('searchUsers')` statement to give an alias for this interception. Notice that we modified the `user.service.ts`, which causes a delay of 5,000 ms before the API call is made.

Cypress has a default timeout of 4,000 ms, and we don't want to restrict our API calls to be handled in our tests within 4,000 ms. Therefore, we use the `cy.wait('@searchUsers');` statement, using the `searchUsers` alias to inform Cypress that it must wait until the aliased interception happens—that is, until the network call is made, regardless of how long it takes until Cypress's second timeout (30,000 ms for network calls) is reached. This makes our current tests pass, even though the default 4,000 ms Cypress timeout and 5,000 ms (about 5 seconds) timeout for HTTP calls in Cypress have already passed before actually making the network call. Magic, isn't it?

Note that Cypress has a default timeout of 4,000 ms for assertions such as checking if an element is visible or has a particular text. It has a default timeout of 5,000 ms for HTTP calls to be initiated. This made our test a bit tricky since we are delaying the initiation of the HTTP call while trying to simulate that the response is delayed from the server. Due to this, we had to use the `options` argument for `cy.wait` including setting the timeout to 10,000 ms. This allows Cypress to wait for the call to be initiated in 5,000 ms (which we added to the user service). In a real-world scenario, your HTTP call will be instantly initiated, and the response may be delayed. Cypress waits for 5,000 ms for the call to be initiated, so you should be fine. As soon as the call is initiated, Cypress defaults to the 30,000 ms timeout to wait for the response.

Well, I hope you liked this recipe—see the next section for a link for further reading.

See also

- Waiting in Cypress: `https://docs.cypress.io/api/commands/wait`

Using Cypress bundled packages

Cypress provides a bunch of bundled tools and packages that we can use in our tests to make things easier, not because writing tests with Cypress is otherwise hard but because these libraries are used by many developers already and so they're familiar with them. In this recipe, we're going to look at the bundled jQuery, Lodash, and Minimatch libraries to test some of our use cases.

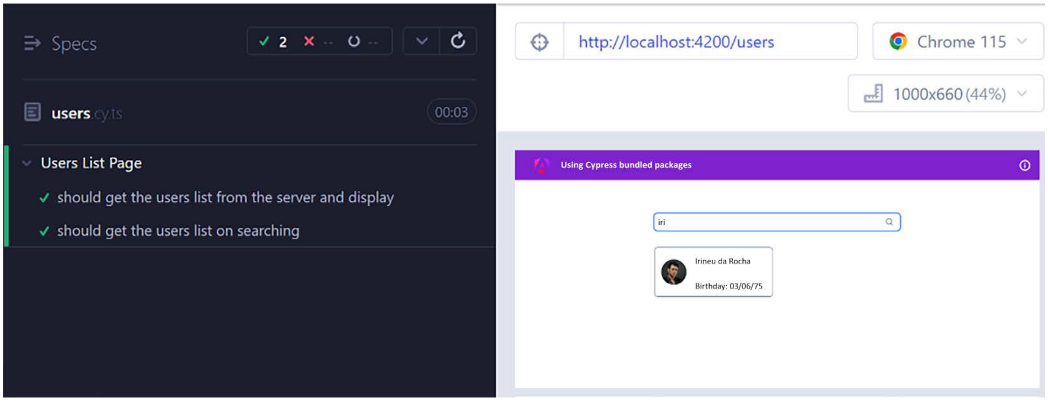
Getting ready

The app that we are going to work with resides in `start/apps/chapter11/ng-cy-bun-pack` inside the cloned repository. However, the e2e tests are in the folder `start/apps/chapter11/ng-cy-bun-pack-e2e`. And in the recipe, we're going to modify files for the e2e project only. Let's run the e2e tests by following these steps:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to run the e2e tests for the project:

```
npm run e2e ng-cy-bun-pack
```

This should open the Cypress window. Select **Chrome** for testing and click the `users.cy.ts` file to run the tests, and you should see the following:



Specs
✓ 2 ✕ -- ○ --
users.cy.ts
00:03
http://localhost:4200/users
Chrome 115
1000x660 (44%)
Using Cypress bundled packages
iri
Irineu da Rocha
Birthday: 03/06/75

Users List Page
- ✓ should get the users list from the server and display
- ✓ should get the users list on searching

Figure 11.13: ng-cy-bun-pack app tests running with Cypress

Now that we have the Cypress tests running, let's see the steps of the recipe in the next section.

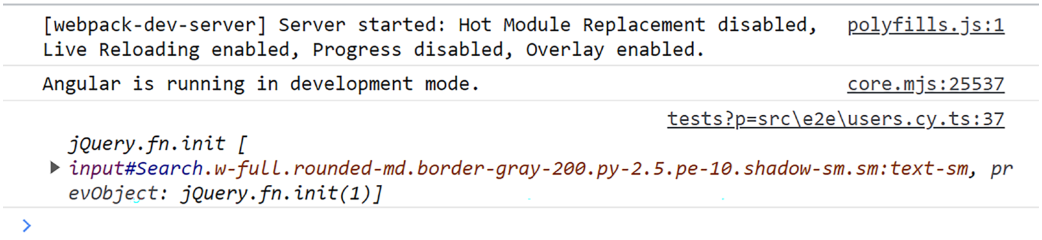
How to do it...

For this recipe, we have the `users` list and a search app that fetches some users from a JSON file using an HTTP request. We're going to assert a few conditions for the DOM, validate the response from the API, and assert the URL changes as well. Let's get started:

1. First, we'll try out the bundled `jQuery` library along with Cypress. We can access this using `Cypress.$`. Let's add another test and log out some DOM elements. Update the `users.cy.ts` file, as follows:

```
...
describe('Users List Page', () => {
  ...
  it('should show the no results found message on search', () => {
    const { $ } = Cypress;
    cy.wait('@searchUsers');
    const searchInput = $('[data-test- id="searchUsersInput"]');
    console.log(searchInput);
  })
});
```

If you look at the tests (Cypress window) now, and specifically the console, you should see the log, as follows:



```
[webpack-dev-server] Server started: Hot Module Replacement disabled,   polyfills.js:1
Live Reloading enabled, Progress disabled, Overlay enabled.
Angular is running in development mode.                                core.mjs:25537
                                                          tests?p=src\e2e\users.cy.ts:37
  jQuery.fn.init [
▶ input#Search.w-full.rounded-md.border-gray-200.py-2.5.pe-10.shadow-sm.sm:text-sm, pr
  evObject: jQuery.fn.init(1)]
>
```

Figure 11.14: Search input logged using jQuery via Cypress.$

2. Now, let's try to change the value of the search input so we can see the `'No results'` message. Update the test further as follows:

```
it('should show the no results found message on search', () => {
    const { $ } = Cypress;
    cy.wait('@searchUsers');
    const searchInput = $('[data-test-id="searchUsersInput"]');
    cy.wrap(searchInput).type('abc123');
});
```

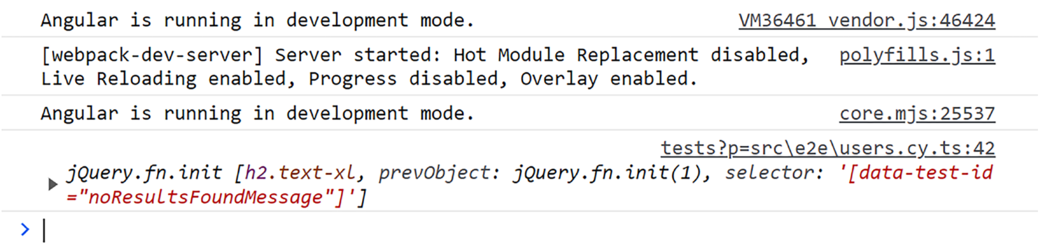
3. Let's add a new page object element in the `users.po.ts` file so we can get the `noResults` message. Update the file as follows:

```
...
export const getNoResultsMessage = () => {
  return cy.get('[data-test-id="noResultsFoundMessage"]');
}
```

4. Let's log the "`no results`" message using the `jQuery` element by using the page object in combination with the `then` method. Update the test in the `users.cy.ts` file as follows:

```
it('should show the no results found message on search', () => {
    const { $ } = Cypress;
    cy.wait('@searchUsers');
    const searchInput = $('[data-test-id="searchUsersInput"]');
    cy.wrap(searchInput).type('abc123');
    getNoResultsMessage().then((el) => {
      console.log(el);
    });
  });
```

You should see the no results message in the console in the Cypress window, as follows:



```
Angular is running in development mode.                        VM36461 vendor.js:46424
[webpack-dev-server] Server started: Hot Module Replacement disabled,   polyfills.js:1
Live Reloading enabled, Progress disabled, Overlay enabled.
Angular is running in development mode.                               core.mjs:25537
                                                         tests?p=src\e2e\users.cy.ts:42
▶ jQuery.fn.init [h2.text-xl, prevObject: jQuery.fn.init(1), selector: '[data-test-id
  ="noResultsFoundMessage"]']
> |
```

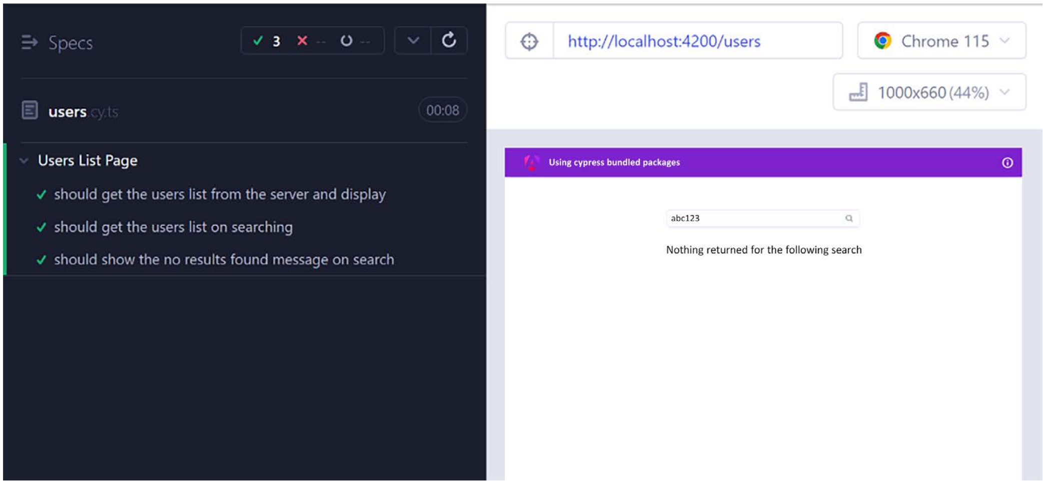
Figure 11.15: noResults message logged using jQuery via Cypress.$

As you can see, the `jQuery` element is logged in the console. We'll now use the Chai assertions to validate that it exists and has a message.

5. Update the test further to check that the element exists and has the following text:

```
it('should show the no results found message on search', () => {
    const { $ } = Cypress;
    cy.wait('@searchUsers');
    const searchInput = $('[data-test-
      id="searchUsersInput"]');
    cy.wrap(searchInput).type('abc123');
    getNoResultsMessage().then((el) => {
      expect(el).to.exist;
      expect(el.text().trim()).to.eq('Nothing returned for
      the following search');
    });
  });
```

If you see the test now in Cypress, it should be passing, as follows:



Users List Page

✓ should get the users list from the server and display

✓ should get the users list on searching

✓ should show the no results found message on search

http://localhost:4200/users

abc123

Nothing returned for the following search

Figure 11.16: Test passing with the usage of jQuery via Cypress

6. We're now going to use the `lodash.js` package that is bundled with Cypress to iterate over each card and ensure the date of birth is correctly formatted. Write another test in the `users.cy.ts` file, as follows:

```
...
describe('Users List Page', () => {
  ...
  it('should show the dob on each user in the correct format', () => {
    const { $, _ } = Cypress;
    cy.wait('@searchUsers');
    getUsersCards().then((cards) => {
      _.forEach(cards, (card) => {
        const cardItem = $(card);
        const dobText = cardItem.find('.dob').text();
        const dob = dobText.split('Birthday:')[1].trim();
        expect(dobRegex.test(
          dob
        )).to.be.true;
      })
    });
  });
});
```

7. Let's add another test to use `lodash` again. We'll ensure that all the names of the users are unique on the view, that is, there are no duplicate user cards. Add another test in the `users.cy.ts` file, as follows:

```
...
describe('Users List Page', () => {
  ...

  it('should have unique names for all the users', () => {
    const { $, _ } = Cypress;
    cy.wait('@searchUsers');
    getUsersCards().then((cards) => {
      const names = _.map(cards, (card) => {
        const cardItem = $(card);
        return cardItem.find('h4').text();
      });
      const uniqueNames = _.uniq(names);
      expect(names.length).to.equal(uniqueNames.length);
    });
  });
});
```

8. The next package we'll explore is the `minimatch` package. When we tap on a user card, it opens the user details. Since we append a timestamp to the URL as a query parameter, we can't compare the URL as an exact match with our assertion. Let's use the `minimatch` package to assert using a pattern instead. Add a new test, as follows:

```
...
describe('Users List Page', () => {
  ...
  it('should go to the user details page with the user uuid', () => {
    const { minimatch, $ } = Cypress;
    getUsersCards().then((cards) => {
      const userCard = cards[0];
      const uuid = $(userCard).attr('ng-reflect-router-link')
      .split('/users/')[1];
      cy.wrap(userCard).click();
      cy.url().should((url) => {
        const urlMatches = minimatch(url,
          `${location.origin}/users/${uuid}*`,
          { debug: true });
          expect(urlMatches).to.equal(true);
        });
      });
    });
  });
```

The tests have now all passed using the bundled packages with Cypress. Now that we've finished the recipe, let's see how it all works in the next section.

How it works...

Cypress bundles `jQuery` with it and we use it via the `Cypress.$` property. This allows us to perform everything that the `jQuery` function permits us to. For example, you can use the following functions from the bundled `jQuery`:

- `each`:
  - Usage: `$(elements).each(function(index, element) {})`
  - Description: Iterates over a jQuery object, executing a function for each matched element
- `text`:
  - Usage: `$(selector).text()`
  - Description: Gets the combined text contents of each element in the set of matched elements, including their descendants
- `val`:
  - Usage: `$(selector).val()`
  - Description: Gets the current value of the first element in the set of matched elements
- `hasClass`:
  - Usage: `$(selector).hasClass(className)`
  - Description: Determines whether any of the matched elements are assigned to the given class
- `addClass`:
  - Usage: `$(selector).addClass(className)`
  - Description: Adds the specified class(es) to each element in the set of matched elements
- `removeClass`:
  - Usage: `$(selector).removeClass(className)`
  - Description: Removes a single class, multiple classes, or all classes from each element in the set of matched elements

IMPORTANT NOTE

`Cypress.$` can only fetch from the document elements that are available immediately on the DOM. This is great for debugging the DOM using the Chrome DevTools in the Cypress test window. However, it is important to understand that it doesn't have any context about the Angular change detection. Also, you can't query any element that isn't visible on the page right from the beginning, as we experienced in the recipe—that is, it doesn't wait for XHR calls for the elements to be visible.

Cypress also bundles `lodash` and exposes it via the `Cypress._` object. In the recipe, we use the `_.each()` method to iterate over the card items to perform several tasks. We also use the `_.uniq` method, which takes an array and returns an array of unique items. We then compare the lengths of both the original array and the unique array to ensure that our original array has all unique names. Note that we can use any `lodash` method in our Cypress tests and not just the mentioned methods.

We also used the `minimatch` package, which Cypress exposes via the `Cypress.minimatch` object. The `minimatch` package is great for matching and testing glob patterns against strings. We use it to test the URL after navigating to a user's detail page using a pattern. One important thing to know when using `minimatch` is that it compares the globs, which should contain the entire URL and not a string *acting like a Regex*. That's why we use the ``` `{location.origin}/users/${uuid}*` ``` statement to include the `location.origin`.

Easy peasy. Now that you understand how the recipe works, see the next section for some useful links.

See also

- Cypress bundled libraries: `https://docs.cypress.io/guides/references/bundled-tools`
- `jQuery`: `https://jquery.com/`
- `lodash`: `https://lodash.com`
- `Minimatch.js`: `https://github.com/isaacs/minimatch`

Using Cypress fixtures to provide mock data

When it comes to writing E2E tests, fixtures play a great role in making sure the tests are not flaky (don't have different results in different test runs). Consider that your tests rely on fetching data from your API server, or your tests include snapshot testing, which includes fetching images from a **Content Delivery Network** (**CDN**) or a third-party API. Although they're technically required for the tests to run successfully, it is not important that the server data and the images are fetched from the original source; therefore, we can create fixtures for them. In this recipe, we'll create fixtures for the bucket data to avoid having to run the server for the e2e tests.

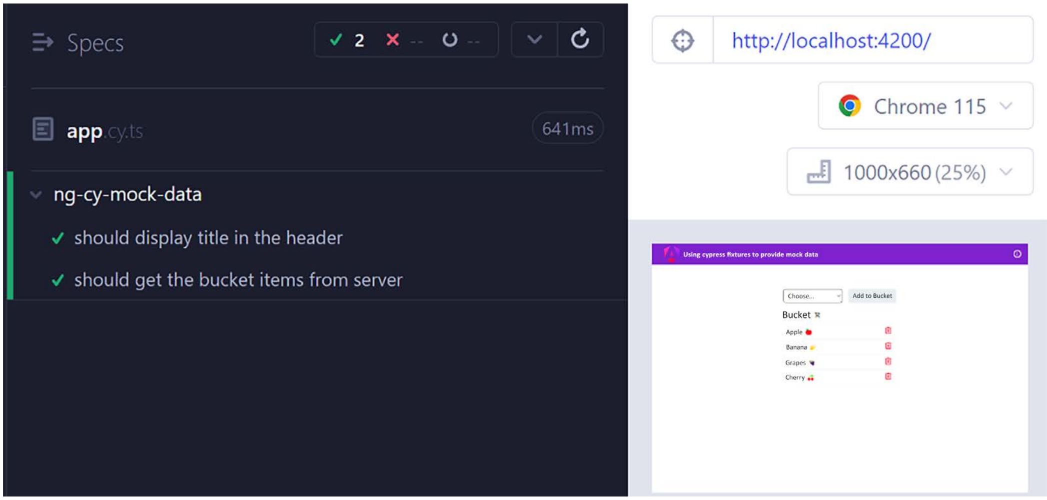
Getting ready

The app that we are going to work with resides in `start/apps/chapter11/ng-cy-mock-data` inside the cloned repository. However, the e2e tests are in the folder `start/apps/chapter11/ng-cy-mock-data-e2e`. In this recipe, we're going to modify files for the e2e project only. Let's run the e2e tests by following these steps:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to run the e2e tests for the project with an API server:

   ```
   npm run e2e ng-cy-mock-data with-server
   ```

   This should open the Cypress window along with the server. Select Chrome for testing and click the `app.cy.ts` file to run the tests, and you should see the following:


Specs
✓ 2 ✕ -- ↻ --
app.cy.ts
641ms
http://localhost:4200/
Chrome 115
1000x660 (25%)
Using cypress fixtures to provide mock data
Choose...
Add to Bucket
Bucket
Apple
Banana
Grapes
Cherry

ng-cy-mock-data
- ✓ should display title in the header
- ✓ should get the bucket items from server

Figure 11.17: ng-cy-mock-data tests running with Cypress

Now that we have the Cypress tests running, let's see the steps of the recipe in the next section.

How to do it...

We have the bucket app, which we have used in many recipes in this book. However, we'll write some e2e tests in this one with Cypress. The interesting part is that the app talks to a backend server to manage the bucket items. Our tests are going to break when we add or remove items from the bucket with the real API. Let's get started:

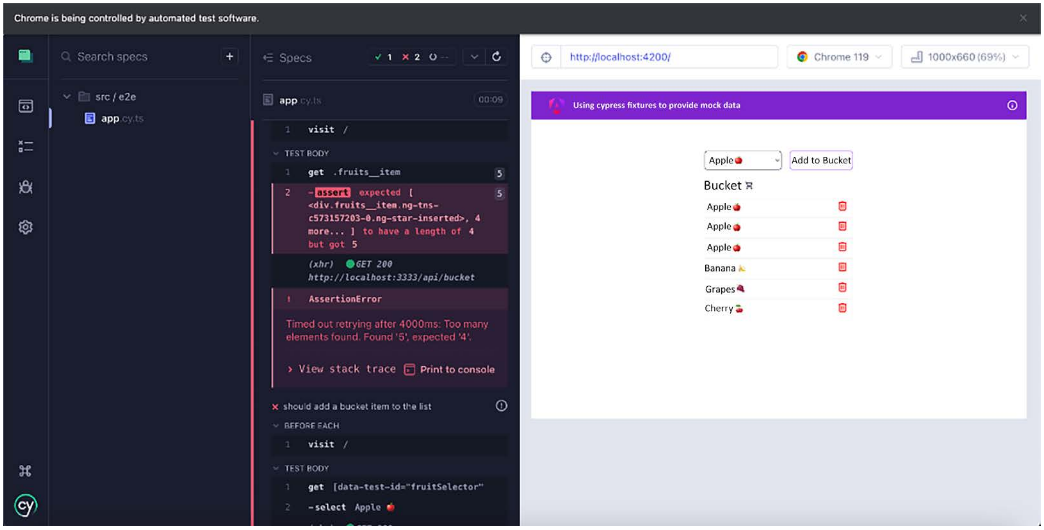
1. Our backend (`fake-be`) returns four bucket items by default. See the *How it works...* section to understand how. We will add a new test to the `app.cy.ts` file later to ensure that we are able to add another item to the bucket. But before that, let's add some page objects in the `app.po.ts` file, as follows:

```
export const getHeaderTitle = () =>
  cy.get('.toolbar__title');
export const getFruits = () => cy.get('.fruits__item');
export const getFruitSelector = () => cy.get('[data-test-
id="fruitSelector"');
export const getAddItemSubmitButton = () =>
  cy.get('[data-test-id="addItemSubmitBtn"');
export const getSuccessToast = () => cy.get('#toast-
container .toast-success');
export const getErrorToast = () => cy.get('#toast-
container .toast-error');
```

2. Now we can add our test to ensure that we're able to add an item to the bucket. Add the following test to the `app.cy.ts` file:

```
import { getAddItemSubmitButton, getFruitSelector, getFruits,
getHeaderTitle, getSuccessToast } from '../support/app.po';

describe('ng-cy-mock-data', () => {
  beforeEach(() => {
    cy.visit('/')
  });


  …


  it('should add a bucket item to the list', () => {
    getFruitSelector().select('Apple 🍏');
    getAddItemSubmitButton().click();
    getSuccessToast().should('be.visible');
    getSuccessToast().then(el => {
      expect(el.text().trim()).to.eq('Bucket item added');
    });
    getFruits().should('have.length', 5);
  });

});
```

This is the point where our tests start to break, as shown in *Figure 11.18*. Since our test adds an item every time it runs to the actual server, we cannot expect the server to return the same number of items (four items) as our test, until we restart the server.



Chrome is being controlled by automated test software.

```
app.cy.ts   00:09
1 visit /
TEST BODY
1 get .fruits__item   5
2 -assert expected [ <div.fruits__item.ng-tns-c573157203-0.ng-star-inserted>, 4 more... ] to have a length of 4 but got 5   5
(xhr) GET 200 http://localhost:3333/api/bucket
! AssertionError
Timed out retrying after 4000ms: Too many elements found. Found '5', expected '4'.
```

× should add a bucket item to the list

```
BEFORE EACH
1 visit /
TEST BODY
1 get [data-test-id="fruitSelector"
2 -select Apple 🍎
```

Using cypress fixtures to provide mock data

Bucket

- Apple 🍎
- Apple 🍎
- Apple 🍎
- Banana 🍌
- Grapes 🍇
- Cherry 🍒

Figure 11.18: Tests failing because of adding data to a real server

3. We will first create a fixture for our HTTP call to the `fake-be` backend. Create a new file under the `src/fixtures` folder, named `get-bucket.json`. Then add the following JSON data to it:

```json
{
  "bucket": [
    { "id": 1, "name": "Apple 🍎" },
    { "id": 2, "name": "Banana 🍌" },
    { "id": 3, "name": "Grapes 🍇" },
    { "id": 4, "name": "Cherry 🍒" }
  ]
}
```

4. Now, let's use the fixture in our `app.cy.ts` file. We'll use it in the `beforeEach` life cycle hook since we want to use the fixture for all the tests in the file. Update the `app.cy.ts` file, as follows:

```typescript
...
describe('ng-cy-mock-data', () => {
  beforeEach(() => {
    cy.fixture("get-bucket")
      .then((response) => {
        cy.intercept('GET', 'http://localhost:3333/api/bucket',
          response)
        return cy.fixture("-bucket");
      })
      .visit('/')
  });
     ...
});
```

This does not fix the issue since the add item to bucket call is still sent to the real API. We'll have to create the fixture for it as well.

5. Create a new file inside the `src/fixtures` folder. Name it `add-bucket-item.json` and add the following code to it:

```
{

        "fruit": { "id": 5, "name": "Apple 🍏" }

}
```

6. We will now use the `add-bucket-item` fixture in our tests file. Update the `app.cy.ts` file, as follows to use the fixture:

```
...

describe('ng-cy-mock-data', () => {
  beforeEach(() => {
    cy.fixture("get-bucket")
      .then((getBucketResp) => {
        cy.intercept('GET', 'http://localhost:3333/api/bucket',
          getBucketResp)
        return cy.fixture("add-bucket-item");
      })
      .then((addItemResp) => {
        cy.intercept('POST',
          'http://localhost:3333/api/bucket', addItemResp)
      })
.visit('/')
  });
  ...
});
```

Now if you run the e2e tests, you should see all the tests passing. It does not matter how many times you refresh the Cypress window; they will always pass because the responses are the same every time:

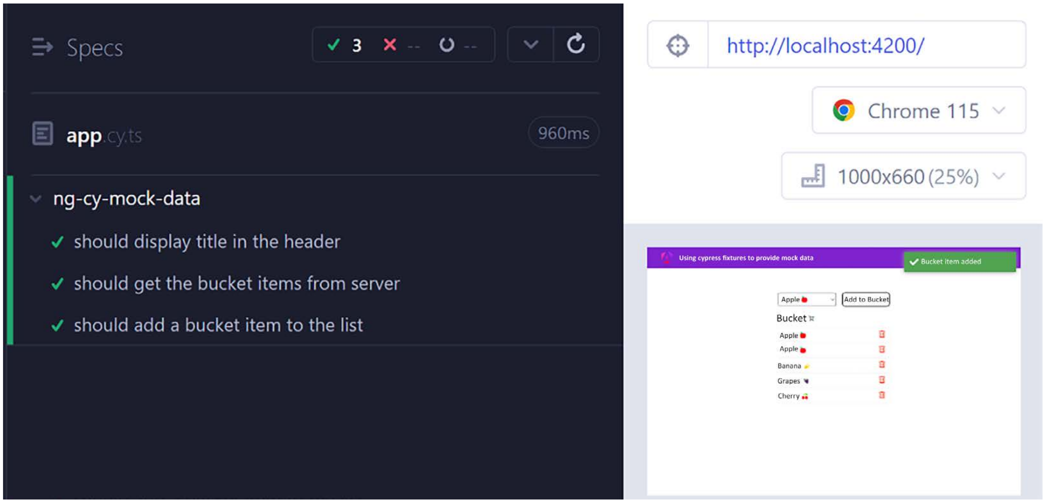
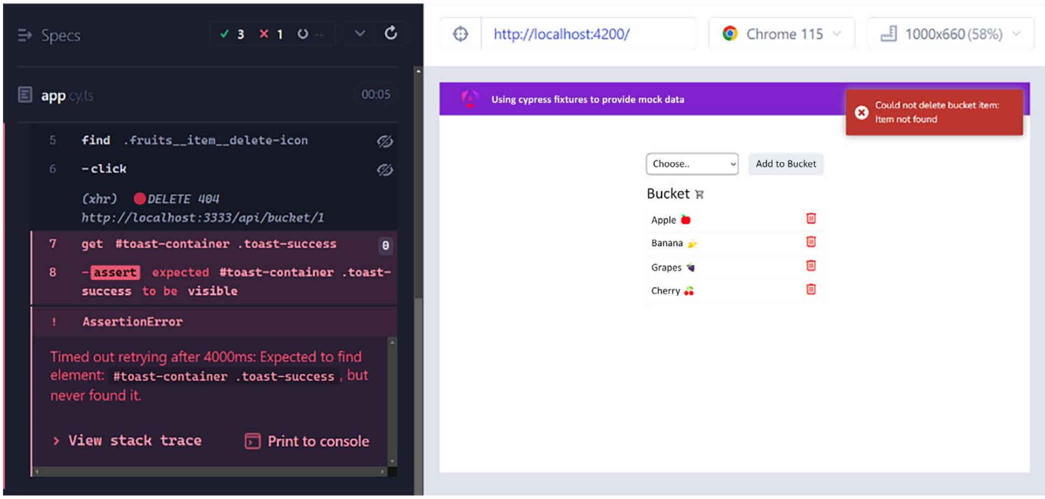

Specs
✓ 3 ✕ -- ↻ --
http://localhost:4200/
Chrome 115
1000x660 (25%)
app.cy.ts
960ms
ng-cy-mock-data
✓ should display title in the header
✓ should get the bucket items from server
✓ should add a bucket item to the list
Using cypress fixtures to provide mock data
Bucket item added
Add to Bucket
Bucket
Apple
Apple
Banana
Grapes
Cherry

Figure 11.19: Tests passing for add-bucket-item with fixtures

7. We will now create a test for deleting an item and ensuring that we see the toast and one item is removed from the view. Let's add another test in the `app.cy.ts` file as follows:

```
...
describe('ng-cy-mock-data', () => {
  ...
  it('should delete a bucket item from the list', () => {
    getFruits().should('have.length', 4);
    getFruits().eq(0).find('.fruits__item__delete-icon').click();
    getSuccessToast().should('be.visible');
    getSuccessToast().then(el => {
      expect(el.text().trim()).to.eq('Bucket item deleted');
    });
    getFruits().should('have.length', 3);
  });
});
```

If you run the tests now, you'll see that the `delete item` test fails because it can't find the item. That's because this `DELETE` call is still being sent to the actual server. If you had your server running, you would see an error, as shown in *Figure 11.20*:


Specs
✓ 3 ✕ 1
app.cy.ts
00:05
http://localhost:4200/
Chrome 115
1000x660 (58%)
Choose..
Add to Bucket
Bucket
Apple
Banana
Grapes
Cherry
View stack trace
Print to console

```
5   find .fruits__item__delete-icon
6   -click
    (xhr) DELETE 404
    http://localhost:3333/api/bucket/1
7   get #toast-container .toast-success   0
8   -assert expected #toast-container .toast-
    success to be visible
!   AssertionError
```

Timed out retrying after 4000ms: Expected to find element: `#toast-container .toast-success`, but never found it.

Using cypress fixtures to provide mock data

Could not delete bucket item: Item not found

Figure 11.20: Item not found error

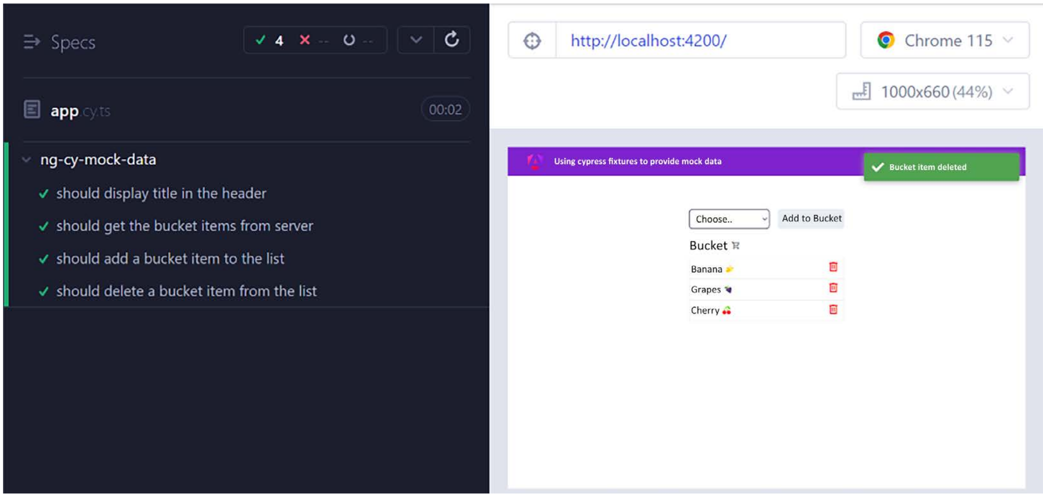
8. We'll create a new fixture in the `src/fixtures` folder named `delete-bucket-item.json`. Add the following code to it:

```
{ "success": true }
```

9. Now let's use the fixture in the `app.cy.ts` file in the `beforeEach()` hook, as follows:

```
...
describe('ng-cy-mock-data', () => {
  beforeEach(() => {
    cy.fixture("get-bucket")
      .then((response) => {
        cy.intercept('GET', 'http://localhost:3333/api/bucket',
          response)
        return cy.fixture("add-bucket-item");
      })
      .then((response) => {
        cy.intercept('POST', 'http://localhost:3333/api/bucket',
          response)
        return cy.fixture("delete-bucket-item");
      })
      .then((deleteItemResp) => {
        cy.intercept('DELETE',
          'http://localhost:3333/api/bucket/*',
          deleteItemResp)
      })
      .visit('/')
  });
  ...
});
```

If you see the tests now, all of them should be passing, as follows:



ng-cy-mock-data
- ✓ should display title in the header
- ✓ should get the bucket items from server
- ✓ should add a bucket item to the list
- ✓ should delete a bucket item from the list

Using cypress fixtures to provide mock data

✓ Bucket item deleted

Bucket
- Banana
- Grapes
- Cherry

Figure 11.21: All tests passing after using fixtures

Great! You now know how to use fixtures in Cypress E2E tests. Now that you've finished the recipe, see the next section for how this works.

How it works...

We had an initial test in the `app.cy.ts` file to ensure that we get four bucket items from the server when the app loads. You can see the backend file that sends the default bucket data, which resides at `<workspace-root>/codewithahsan/e2e/fake-be/src/app/bucket/bucket.service.ts`. However, as soon as we start adding items to the bucket via our tests, our tests break because we're working with real data. And we should rarely be doing that to ensure we can make assertions accurately. That's why big teams have test environments where they seed the data into the database if they really want to work with real data. Since our bucket app is super small, and we don't really need to work with real data, we add fixtures in this recipe. Fixtures in a Cypress test are registered with the `cy.fixture` method, which allows us to use data from a file. In this recipe, we use fixtures for all the HTTP calls the app makes to the `fake-be` server, that is, the following:

- Get all bucket data – `GET` `http://localhost:3333/api/bucket`
- Add item to the bucket – `POST` `http://localhost:3333/api/bucket`
- Delete item from the bucket – `DELETE` `http://localhost:3333/api/bucket/ITEM_ID`

Notice that for each HTTP call, we use the `cy.fixture('FIXTURE_NAME')` without the `.json` extension, which essentially points to the `cypress/fixture/FIXTURE_NAME.json` file.

First, we use the `cy.fixture` method to register the fixture (or to fetch it). Then we use the `then` method to get the contents of the fixture (JSON) file. Then we use the `cy.intercept` method to intercept the HTTP call using the `GET`/`POST`/`DELETE` method and the URL pattern as a `Minimatch` glob pattern to intercept the HTTP call to get the fixture response and provide it as the response for the HTTP call. As a result, all the intercepted calls matching the glob patterns use our fixtures.

Now that you understand how the recipe works, see the next section for some resources.

See also

- Cypress fixtures documentation: `https://docs.cypress.io/api/commands/fixture`
- `cy.intercept` method documentation: `https://docs.cypress.io/api/commands/intercept`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

12

Performance Optimization in Angular

Performance is always a concern in any product that you build for end users. It is a critical element in increasing the chances of someone using your app for the first time becoming a customer. Now, we can't really improve an app's performance until we identify potential possibilities for improvement and the methods to achieve this. In this chapter, you'll learn some methods to deploy when it comes to improving Angular applications. You'll learn how to analyze, optimize, and improve your Angular app's performance using several techniques. Here are the recipes we're going to cover in this chapter:

- Using `OnPush` change detection to prune component subtrees
- Detaching the change detector from components
- Running `async` events outside Angular with `runOutsideAngular`
- Using `trackBy` for lists with `*ngFor`
- Moving heavy computation to pure pipes
- Using web workers for heavy computation
- Using performance budgets for auditing
- Analyzing bundles with `webpack-bundle-analyzer`

Technical requirements

For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter12`.

Using OnPush change detection to prune component subtrees

In today's world of modern web applications, performance is one of the key factors for a great **User Experience** (UX) and, ultimately, conversions for a business. In this recipe, the first recipe of this chapter, we're going to discuss the fundamental or the most basic optimization you can do with your components wherever it seems appropriate, which is by using the `OnPush` change detection strategy. The app we're working with has some performance issues, particularly with the `UserCardComponent` class. This is because it uses a getter function, `randomColor`, to generate a random color for its background. Behind the scenes, that function uses a `factorial` function to add more processing time. But that is just to demonstrate a component that can cause the UI to hang if there is some complex calculation happening, along with multiple change detections being triggered.

Getting ready

The app that we are going to work with resides in `start/apps/chapter12/ng-on-push-strategy` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-on-push-strategy
```

This should open the app in a new browser tab, and you should see the following:

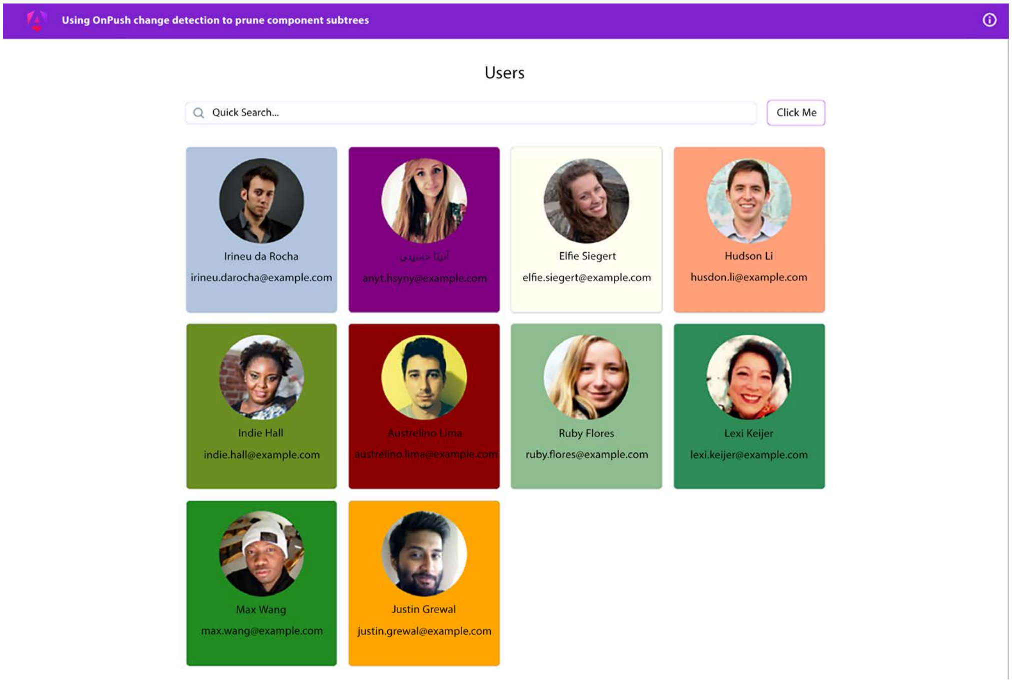

Using OnPush change detection to prune component subtrees
Users
Quick Search...
Click Me
Irineu da Rocha
irineu.darocha@example.com
آنیتا حسینی
anyt.hsyny@example.com
Elfie Siegert
elfie.siegert@example.com
Hudson Li
husdon.li@example.com
Indie Hall
indie.hall@example.com
Austrelino Lima
austrelino.lima@example.com
Ruby Flores
ruby.flores@example.com
Lexi Keijer
lexi.keijer@example.com
Max Wang
max.wang@example.com
Justin Grewal
justin.grewal@example.com

Figure 12.1: The ng-on-push-strategy app running at http://localhost:4200

Click the button that says **Click Me** or try to search for some user. You'll see that the app is too slow and often hangs. Now that we have the project served on the browser, let's see the steps of the recipe in the next section.

How to do it...

We're going to add some code to monitor how many times the `randomColor` getter gets called. This will show the number of times the change detection in Angular gets triggered by default. We will also fix the issue and make it more performant (as much as we can) using the `OnPush` change detection strategy. Let's get started:

1. First, let's make sure that the app is not too slow for your machine so that it makes your laptop/PC hang. Open the `src/app/app.config.ts` file and adjust the value of the `RANDOMIZATION_COUNT` token from `9` to whatever suits you best.
2. Then try to search for a user named `Irineu` by entering their name in the search box. You'll notice that the app still hangs and that it takes a few seconds to show the user. You'll also notice that you don't even see the typed letters in the search box as you type them. That is, there is a delay in rendering.

   Let's add some logic to the code. We'll check how many times Angular calls the `idUsingFactorial` method when the page loads.

3. Let's create a service that we will use to track the number of times a particular user card for a particular person is called. Run the following command from the workspace's root folder to create a service:

   ```
   cd start && nx g s services/logs --project ng-on-push-strategy
   ```

   Choose `@schematics/angular:service` when asked.

4. Update the content of the `src/app/services/logs.service.ts` file as follows:

   ```
   import { Injectable } from '@angular/core';

   @Injectable({
     providedIn: 'root'
   })
   export class LogsService {
     logs: Record<string, number> = {}

     updateLogEntry(email: string) {
       if (this.logs[email] === undefined) {
         this.logs = {
           ...this.logs,
           [email]: 1
         }
   ```

```
      } else {
        this.logs = {
          ...this.logs,
          [email]: this.logs[email] + 1
        }
      }
    }
  }
```

5. Now, inject the `LogService` in the `src/app/component/user-card/user-card.component.ts` file. We will also create a getter (`log`) function to get the count for the user, and we will update the count whenever the `randomColor` getter is called. Update the mentioned file as follows:

```
...
import { LogsService } from '../../services/logs.service';

@Component({...})
export class UserCardComponent {
  ...
  randomizationCount = inject(RANDOMIZATION_COUNT);
  logsService = inject(LogsService);

  get log() {
    return this.logsService.logs[this.user.email] ?? 0;
  }

  get randomColor() {
    this.logsService.updateLogEntry(this.user.email);
    ...
  }
}
```

6. Now, we will use the log in the template of the user card component to show the count. Update the `src/app/component/user-card/user-card.component.html` file as follows:

```
<div [style.backgroundColor]="randomColor"...>
  <img ...>
  <div class="card-body flex-1">...</div>
  <div class="p-4 bg-slate-900 text-green-300 rounded-md h-fit">
    <div>
      Color Generation Count:
    </div>
```

```
      <pre>{{log}}</pre>
    </div>
  </div>
```

If you look at the application now, you should see the **Color Generation Count** as follows:

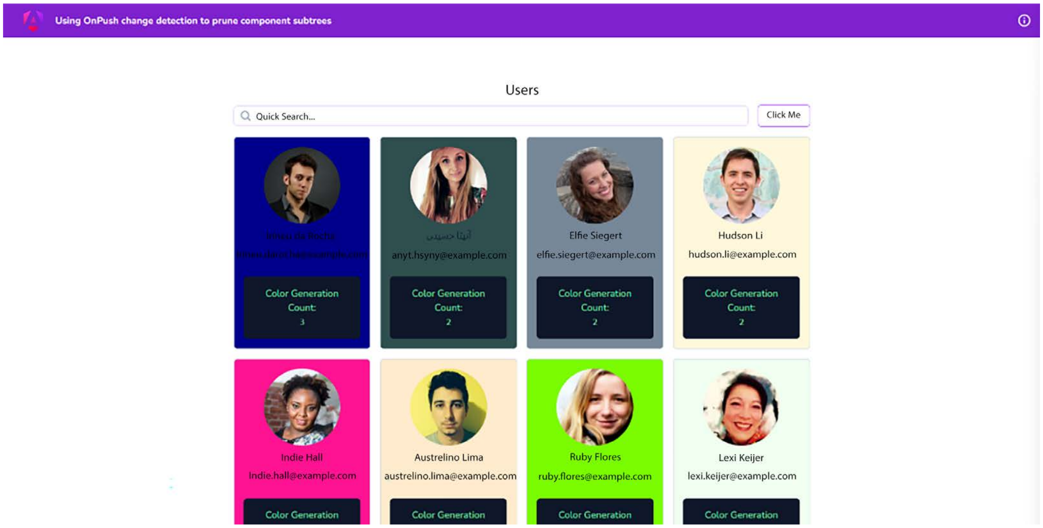

Using OnPush change detection to prune component subtrees
Users
Quick Search...
Click Me
Irineu da Rocha
irineu.darocha@example.com
Color Generation Count: 3
أنيتا حسيني
anyt.hsyny@example.com
Color Generation Count: 2
Elfie Siegert
elfie.siegert@example.com
Color Generation Count: 2
Hudson Li
hudson.li@example.com
Color Generation Count: 2
Indie Hall
indie.hall@example.com
Color Generation
Austrelino Lima
austrelino.lima@example.com
Color Generation
Ruby Flores
ruby.flores@example.com
Color Generation
Lexi Keijer
lexi.keijer@example.com
Color Generation

Figure 12.2: Color Generation Count being shown on page load

7. Now, click the **Click Me** button. Then, focus (click) the **Quick Search** input and then click outside. Repeat this a few times and you should see that the colors are regenerated even though the cards are not supposed to be re-rendered. *Figure 12.3* shows how it should look:

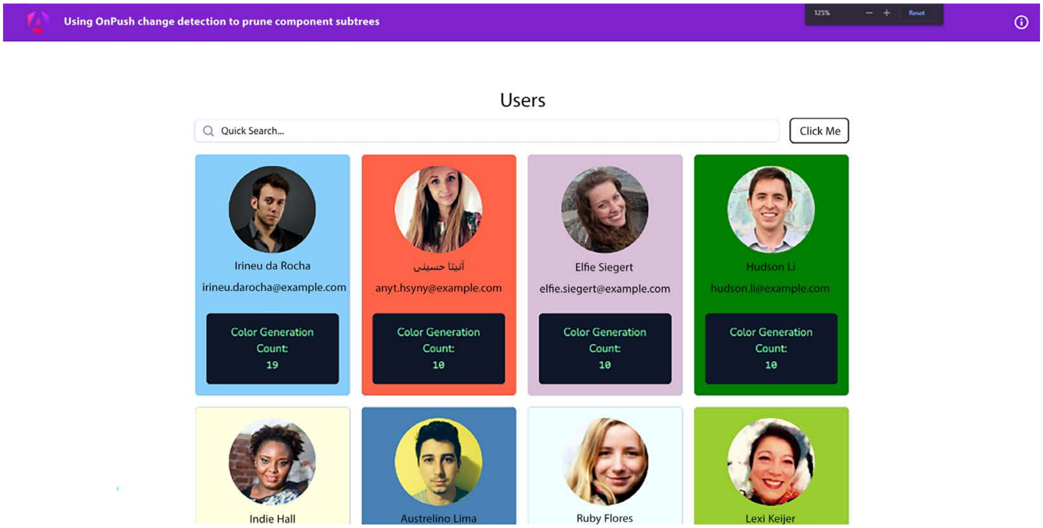

Using OnPush change detection to prune component subtrees
125%
Reset
Users
Quick Search...
Click Me
Irineu da Rocha
irineu.darocha@example.com
Color Generation Count: 19
أنيتا حسيني
anyt.hsyny@example.com
Color Generation Count: 10
Elfie Siegert
elfie.siegert@example.com
Color Generation Count: 10
Hudson Li
hudson.li@example.com
Color Generation Count: 10
Indie Hall
Austrelino Lima
Ruby Flores
Lexi Keijer

Figure 12.3: Logs after interacting with the app without searching for anything

Notice that if you start searching for something, you'll get even more re-renders. That's because each key up and/or key down event will trigger more re-renders.

8. To fix the issue, we will use the `OnPush` strategy and see how it changes the behavior of the user card component with respect to Angular's change detection strategy. Update the `user-card.component.ts` file as follows:

```
import { ChangeDetectionStrategy, Component, Input, inject } from '@angular/core';
...
@Component({
  ...,
  styleUrls: ['./user-card.component.scss'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
...
```

Now, if you try clicking the **Click Me** button, focus in the search input and click outside, or do whatever else (apart from searching for users), you will see no change in the **Color Generation Count** for the cards, as shown in *Figure 12.4*:

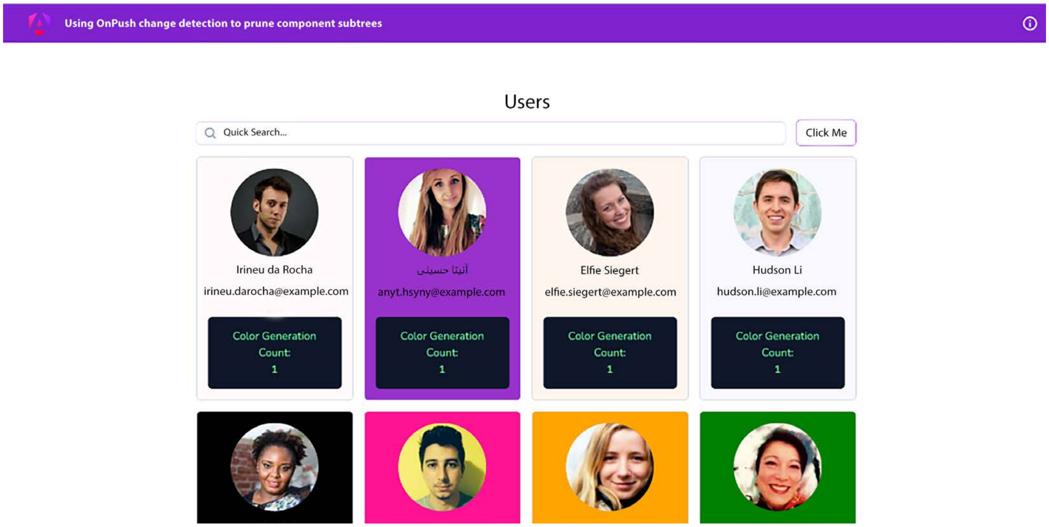

Using OnPush change detection to prune component subtrees
Users
Quick Search...
Click Me
Irineu da Rocha
irineu.darocha@example.com
Color Generation Count: 1
أنيتا حسيني
anyt.hsyny@example.com
Color Generation Count: 1
Elfie Siegert
elfie.siegert@example.com
Color Generation Count: 1
Hudson Li
hudson.li@example.com
Color Generation Count: 1

Figure 12.4: The OnPush strategy preventing unnecessary renders

Great! By using the `OnPush` strategy, we were able to improve the overall performance of our `UserCardComponent`. The rest was just for fun. Now you know how to use this strategy, see the next section to understand how it works.

How it works...

Angular by default uses the `Default` change detection strategy—or technically, it is the `ChangeDetectionStrategy.Default` enum from the `@angular/core` package. Since Angular doesn't know about every component we create, it uses the default strategy to not encounter any surprises. This means that the framework will check for changes in the whole component tree whenever something might have changed. This might include user events, timers, XHRs, promises, etc. In a complex UI with lots of bindings, this can lead to performance degradation, especially when most of these components don't change frequently or depend only on specific inputs.

But as developers, if we know that a component will not change unless one of its `@Input()` variables changes, we can—and we should—use the `OnPush` change detection strategy for that component. Why? Because it tells Angular to not run change detection until an `@Input()` variable for the component changes. This strategy is an absolute winner for presentational components (sometimes called `dumb` components), which are just supposed to show data using `@Input()` variables/attributes and emit `@Output()` events on interactions. These presentational components usually do not hold any business logic such as heavy computation, using services to make **HTTP** calls, and so on. Therefore, it is easier for us to use the `OnPush` strategy in these components because they would only show different data when any of the `@Input()` attributes from the parent component change, that is, their reference should change. For example, if we had an array of users, there should be a completely new array now in order to run the change detection. If it is the same array but we have only added or removed an item, change detection with `OnPush` will not be triggered.

Since we are now using the `OnPush` strategy on our `UserCardComponent`, it only triggers change detection when we replace the entire `users` array upon searching. This happens after the `500ms` debounce (*line 31* in the `users.component.ts` file), so we only do it when the user stops typing. So, essentially, before the optimization, the default change detection was triggering on each keypress being a browser event, and now, it doesn't.

> IMPORTANT NOTE
>
> As you now know, the `OnPush` strategy only triggers the Angular change detection mechanism when one or more of the `@Input()` bindings changes. This means that if we change a property within the component (`UserCardComponent`), it will not be reflected in the view because the change detection mechanism won't run in this case, since that property isn't an `@Input()` binding. You would have to mark the component as dirty so that Angular could check the component and run change detection. You'll do this using the `ChangeDetectorRef` service—specifically, with the `markForCheck` method.

See also

- Angular `ChangeDetectionStrategy` official documentation: `https://angular.io/api/core/ChangeDetectionStrategy`
- The `markForCheck` method official documentation: `https://angular.io/api/core/ChangeDetectorRef#markforcheck`

Detaching the change detector from components

In the previous recipe, we learned how to use the `OnPush` strategy in our components to avoid Angular change detection running unless one of the `@Input()` bindings has changed. There is, however, another way to tell Angular to not run change detection at all for a particular component and its subtree. This cuts the component and its subtree from the change detection cycle completely, as shown in *Figure 12.5*, which can result in an increase in the overall performance. This is also handy when you want full control of when to run change detection. In this recipe, you'll learn how to completely detach the change detector from an Angular component to gain performance improvements.

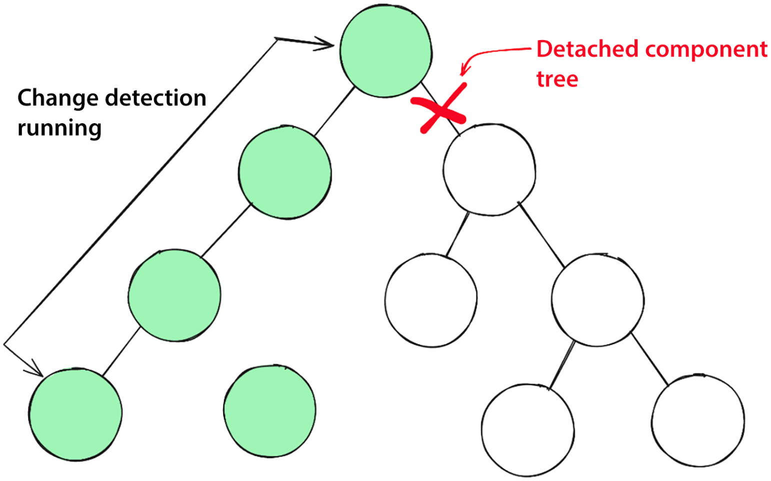


Figure 12.5: Change detector detached from component tree

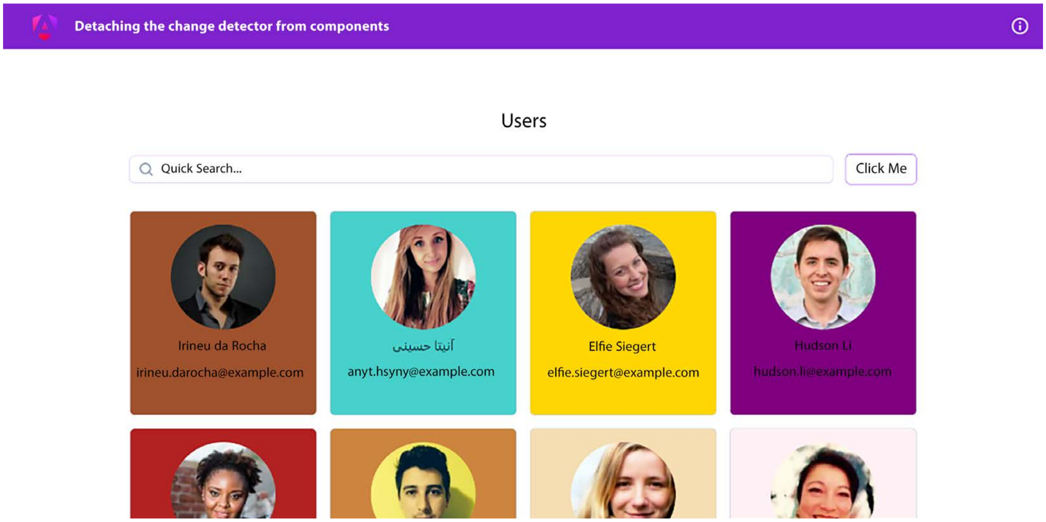
Getting ready

The app that we are going to work with resides in `start/apps/chapter12/ng-cd-ref` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

   ```
   npm run serve ng-cd-ref
   ```

   This should open the app in a new browser tab, and you should see the following:



Detaching the change detector from components

Users

Irineu da Rocha
irineu.darocha@example.com

أنيتا حسيني
anyt.hsyny@example.com

Elfie Siegert
elfie.siegert@example.com

Hudson Li
hudson.li@example.com

Figure 12.6: The ng-cd-ref app running at http://localhost:4200

Click the button that says **Click Me** Or try to search for some user. You'll see that the app is too slow and often hangs. Now that we have the project served on the browser, let's see the steps of the recipe in the next section.

How to do it...

The app we're working with has some performance issues, particularly with the `UserCardComponent` class. This is because it uses a getter function, `randomColor`, to generate a random color for its background. Behind the scenes, that function uses a `factorial` function to add more processing time. But that is just to demonstrate a component that can cause the UI to hang if there is some complex calculation happening along with multiple change detections being triggered. We're going to add some code to monitor how many times the `randomColor` getter gets called. This will show the number of times the change detection in Angular gets triggered by default. We will also fix the issue and make it more performant (as much as we can) by detaching the change detector completely from a particular component. Let's get started:

1. First, let's make sure that the app is not too slow for your machine so that it makes your laptop/PC hang. Open the `src/app/app.config.ts` file and adjust the value of the `RANDOMIZATION_COUNT` token from `9` to whatever suits you best.
2. Then try to search for a user named `Irineu` by entering their name in the search box. You'll notice that the app still hangs and that it takes a few seconds to show the user. You'll also notice that you don't even see the letters in the search box as you type them. That is, there is a delay in rendering.

   Let's add some logic to the code. We'll check how many times Angular calls the `idUsingFactorial` method when the page loads.

3. Let's create a service that we will use to track the number of times a particular user card for a particular person is called. Run the following command from the workspace's root folder to create a service:

   ```
   cd start && nx g s services/logs --project ng-cd-ref
   ```

   Choose `@schematics/angular:service` when asked.

4. Update the content of the `src/app/services/logs.service.ts` file as follows:

   ```
   import { Injectable } from '@angular/core';

   @Injectable({
     providedIn: 'root'
   })
   export class LogsService {
     logs: Record<string, number> = {}

     updateLogEntry(email: string) {
       if (this.logs[email] === undefined) {
         this.logs = {
           ...this.logs,
           [email]: 1
         }
       } else {
         this.logs = {
           ...this.logs,
           [email]: this.logs[email] + 1
         }
       }
     }
   }
   ```

5. Now, inject the `LogService` in the `src/app/component/user-card/user-card.component.ts` file. We will also create a getter (`log`) function to get the count for the user, and we will update the count whenever the `randomColor` getter is called. Update the mentioned file as follows:

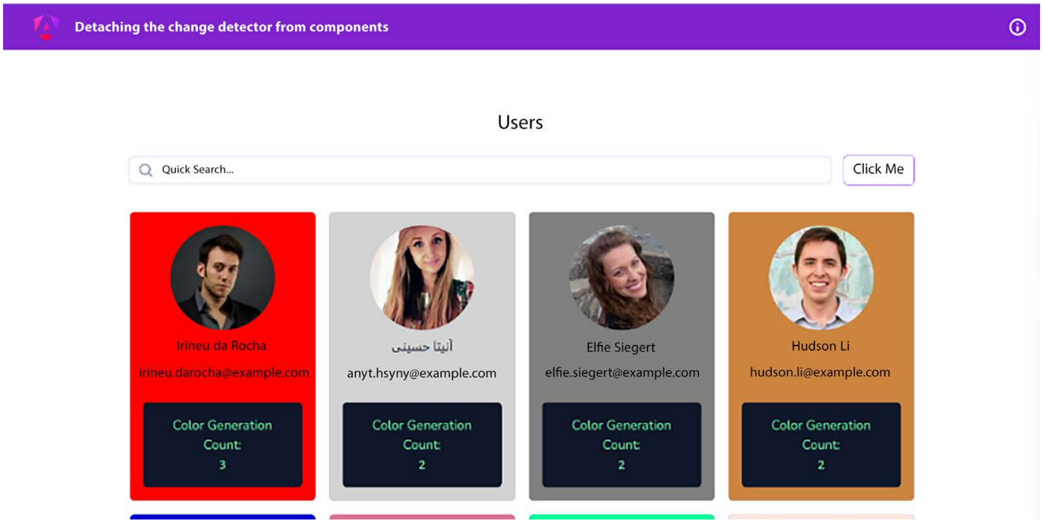
   ```
   ...
   import { LogsService } from '../../services/logs.service';

   @Component({...})
   export class UserCardComponent {
     ...
     randomizationCount = inject(RANDOMIZATION_COUNT);
     logsService = inject(LogsService);
   ```

```
  get log() {
    return this.logsService.logs[this.user.email] ?? 0;
  }

  get randomColor() {
    this.logsService.updateLogEntry(this.user.email);
    ...
  }
}
```

6. Now, we will use the log in the template of the user card component to show the count. Update the `src/app/component/user-card/user-card.component.html` file as follows:

```
<div [style.backgroundColor]="randomColor"...>
  <img ...>
  <div class="card-body flex-1">...</div>
  <div class="p-4 bg-slate-900 text-green-300 rounded-md h-fit">
    <div>
      Color Generation Count:
    </div>
    <pre>{{log}}</pre>
  </div>
</div>
```

If you look at the application now, you should see the **Color Generation Count** as follows:



Irineu da Rocha
irineu.darocha@example.com
Color Generation Count: 3

أنيتا حسينى
anyt.hsyny@example.com
Color Generation Count: 2

Elfie Siegert
elfie.siegert@example.com
Color Generation Count: 2

Hudson Li
hudson.li@example.com
Color Generation Count: 2

Figure 12.7: Color Generation Count being shown on page load

7. Now, click the **Click Me** button. Then focus (click) the **Quick Search** input and then click outside. Repeat this a few times and you should see that the colors are regenerated even though the cards are not supposed to be re-rendered. *Figure 12.8* shows how it should look:

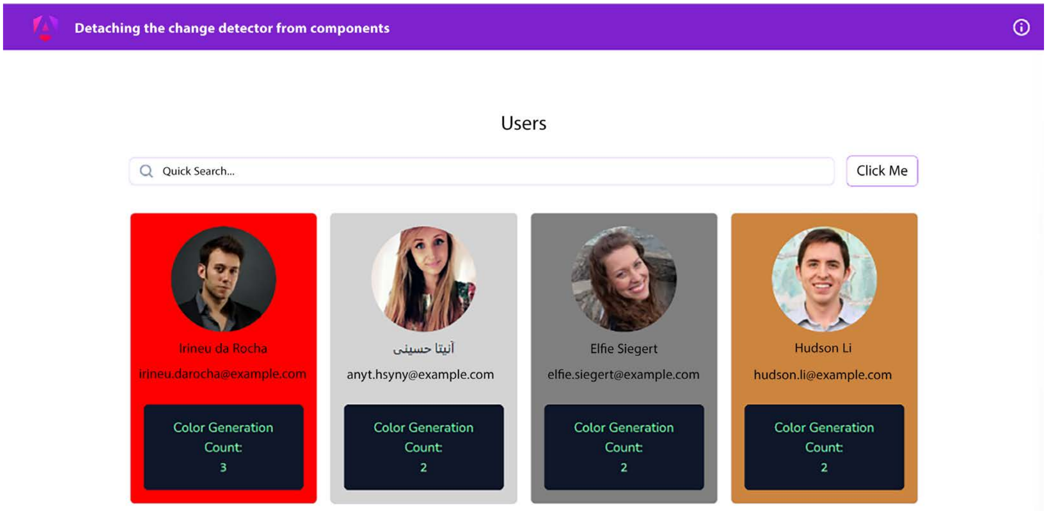

Detaching the change detector from components
Users
Quick Search...
Click Me
Irineu da Rocha
irineu.darocha@example.com
Color Generation Count: 3
آنیتا حسینی
anyt.hsyny@example.com
Color Generation Count: 2
Elfie Siegert
elfie.siegert@example.com
Color Generation Count: 2
Hudson Li
hudson.li@example.com
Color Generation Count: 2

Figure 12.8: Logs after interacting with the app without searching for anything

Notice that if you start searching for something, you'll get even more re-renders. That's because each key up and/or key down event will trigger more re-renders.

8. To fix the issue, we will detach the change detector reference from the `user-card` components when the component loads so there is no re-render afterward. This is assuming that the content of the card never changes. Update the `user-card.component.ts` file as follows:

```
import { AfterViewInit, ChangeDetectorRef, Component, Input, inject }
from '@angular/core';
      ...

export class UserCardComponent implements AfterViewInit {
  ...
  logsService = inject(LogsService);
  cdRef = inject(ChangeDetectorRef);

  ngAfterViewInit(): void {
    this.cdRef.detach();
  }
  ...
}
```

Now, if you try clicking the **Click Me** button, focus on the search input and click outside, or do whatever else (apart from searching for users), you will see no change in the **Color Generation Count** for the cards, as shown in *Figure 12.9*:

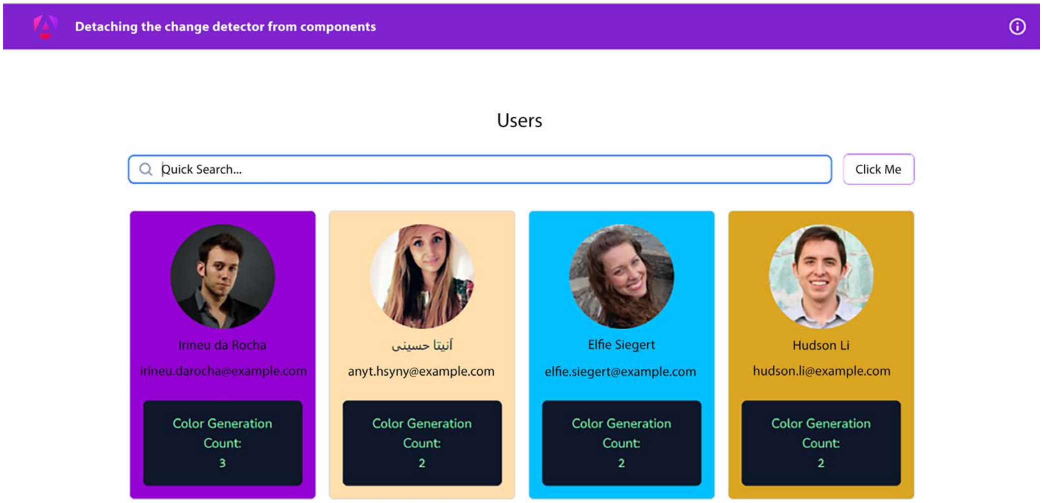

Detaching the change detector from components
Users
Quick Search...
Click Me
Irineu da Rocha
irineu.darocha@example.com
Color Generation Count: 3
أنيتا حسینی
anyt.hsyny@example.com
Color Generation Count: 2
Elfie Siegert
elfie.siegert@example.com
Color Generation Count: 2
Hudson Li
hudson.li@example.com
Color Generation Count: 2

Figure 12.9: Detached change detector preventing unnecessary renders

But what if something had to change in the components later? How do we get full control over the change detection?

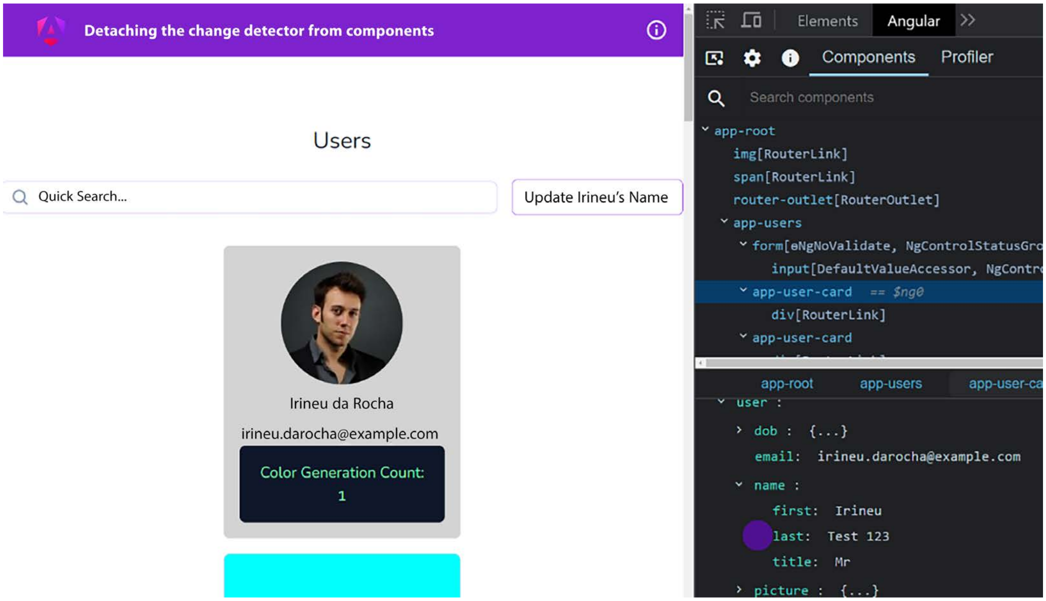
9. Let's assume that the app has a feature that changes the user's name. We'll hardcode the logic to change the last name of the first user. In this case, we would have to tell Angular to run the change detection. Let's update the `users.component.html` file to update the **Click Me** button in the `src/app/users/users.component.html` file as follows:

```
<div class="home">
  <section class="flex flex-col gap-4 w-full">
    <h2 class="text-center text-2xl">Users</h2>
    <form class="input-container flex gap-4 w-full items
      -center mb-4" [formGroup]="searchForm">
      <div class="relative flex-1">...</div>
      <button (click)="updateName(users[0])">Update
        Irineu's Name</button>
    </form>
    <div class="secondary-container flex justify-center">...</div>
  </section>
</div>

<ng-template #loader>
  <app-loader></app-loader>
</ng-template>
```

10. Let's update the TypeScript file to add the function that updates the user's name. Update the `src/app/users/users.component.ts` file as follows:

```
    ...
export class UsersComponent implements OnInit {
  ...

  usersTrackBy(_index: number, user: IUser) {
    return user.uuid;
  }

  updateName(user: IUser) {
    this.users = this.users.map((userItem) => {
      if (userItem.uuid === user.uuid) {
        return {
          ...userItem,
          name: {
            ...userItem.name,
            last: 'Test 123'
          }
        }
      }
      return userItem;
    })
  }
}
```

If you click the **Update Irineu's Name** button, you'll see no change in the UI. That is because the change detector is still detached from each user card component. So, the name changes, but you can't see the change on the UI. See *Figure 12.10*, in which the Angular (Chrome) DevTools show the value being updated in the component, but the UI doesn't reflect it.


Detaching the change detector from components
Users
Quick Search...
Update Irineu's Name
Irineu da Rocha
irineu.darocha@example.com
Color Generation Count:
1
Elements
Angular
Components
Profiler
Search components
app-root
app-users
app-user-ca

```
˅ app-root
    img[RouterLink]
    span[RouterLink]
    router-outlet[RouterOutlet]
  ˅ app-users
    ˅ form[ɵNgNoValidate, NgControlStatusGro
        input[DefaultValueAccessor, NgContro
    ˅ app-user-card  == $ng0
        div[RouterLink]
    ˅ app-user-card
```

```
˅ user :
  › dob :  {...}
    email:  irineu.darocha@example.com
  ˅ name :
      first:  Irineu
      last:  Test 123
      title:  Mr
  › picture :  {...}
```

Figure 12.10: User card not re-rendering due to detached change detector

11. To fix the issue, we will query the `UserCard` components on the users page and will manually run the `ChangeDetectorRef.detectChanges` method on the desired component. Update the `src/app/users/users.component.ts` file as follows:

```
import { Component, inject, OnInit, QueryList, ViewChildren } from '@
angular/core';
...
export class UsersComponent implements OnInit {
  @ViewChildren(UserCardComponent) userCards!:
    QueryList<UserCardComponent>;
  users!: IUser[];
  ...

  updateName(user: IUser) {
    this.users = this.users.map((userItem) => {
      ...
    })
```

```
        const matchingComponent = this.userCards.find(comp => {
          return comp.user.uuid === user.uuid;
        })
        if (matchingComponent) {
          setTimeout(() => {
            matchingComponent.cdRef.detectChanges();
          }, 0);
        }
      }
    }
```

Now, if you click the **Update Irineu's Name** button four times, you should see the count of the first user card to be `5`, while the rest of the cards still render `1` time, as shown in *Figure 12.11*.

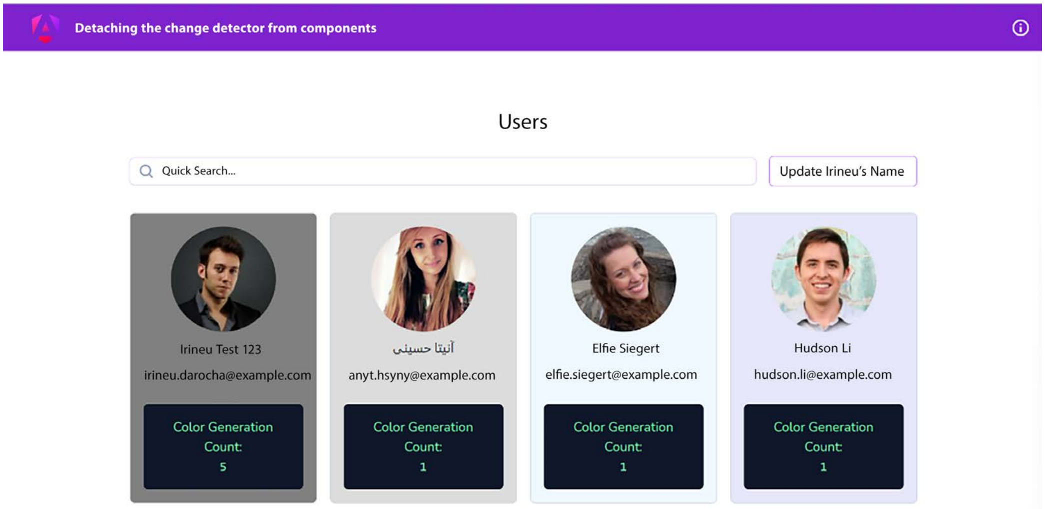

Detaching the change detector from components
Users
Quick Search...
Update Irineu's Name
Irineu Test 123
irineu.darocha@example.com
Color Generation Count: 5
أنيتا حسينى
anyt.hsyny@example.com
Color Generation Count: 1
Elfie Siegert
elfie.siegert@example.com
Color Generation Count: 1
Hudson Li
hudson.li@example.com
Color Generation Count: 1

Figure 12.11: Fully controlled change detection

Great! Within a few steps, by using the `ChangeDetectorRef` service from Angular, we were able to improve the overall performance of our `UserCardComponent`. Not only did we improve it, but we also fully controlled it from the parent component (`UsersComponent`) according to our use case. Now you know how to use the `ChangeDetectorRef` service, see the next section to understand how it works.

How it works...

The `ChangeDetectorRef` service provides a range of important methods to control change detection in Angular. In the recipe, we use the `detach` method in our component's (`UserCardComponent` class's) `ngAfterViewInit` method to detach the Angular change detection mechanism from the component as soon as it is rendered for the first time. As a result, no change detection is triggered on the `UserCardComponent` class. This is because Angular has a change detection tree in which each component is a node. When we detach a component from the change detection tree, that component (as a tree node) is detached, and so are its child components (or nodes). By doing this, we end up with absolutely no change detection happening for the `UserCardComponent` class. If there were any other components being used in the `UserCardComponent`, they would not have the change detection run for them either. As a result, when we click the button or focus and blur the input on the users page, nothing is rendered, even after we update the first user's name as we did in the recipe.

Furthermore, when we need to show the changed name of the first user on the view, which requires the Angular change detection mechanism to be triggered, we use the `detectChanges` method from the `ChangeDetectorRef` instance, right after we've assigned the updated users array to the `users` property in the `UsersComponent` class. As a result, Angular runs the change detection mechanism, and we see the updated name on the first user card. This gives us the ultimate power to decide whether we want to detach the change detection completely, reattach it, or manually run it for the only cases that require change detection.

Now that you understand how the recipe works, see the next section for some useful links.

See also

- The `ChangeDetectorRef` official documentation: `https://angular.io/api/core/ChangeDetectorRef`

Running async events outside Angular with runOutsideAngular

Angular runs its change detection mechanism on a couple of things, including—but not limited to—all browser events, such as `keyup`, `keydown`, `click`, and so on. It also runs change detection on `setTimeout`, `setInterval`, and Ajax HTTP calls. If we had to avoid running change detection on any of these events, we'd have to tell Angular not to trigger change detection on them—for example, if you use the `setInterval` method in your Angular component, it will trigger an Angular change detection cycle each time its callback method is called. This can lead to a huge amount of change detection cycles and your app might even hang. An ideal situation would be to be able to still use the `setInterval` method and so on without triggering the change detection. In this recipe, you'll learn how to do exactly that. You will learn how to execute code blocks outside of `zone.js` using the `NgZone` service, particularly using the `runOutsideAngular` method. See *Figure 12.12* to understand the structure of the application:

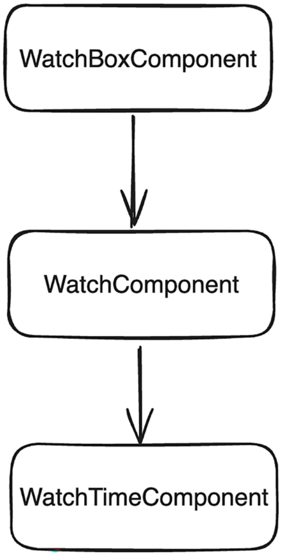


Figure 12.12: Component hierarchy for the ng-run-outside-angular app

Getting ready

The app that we are going to work with resides in `start/apps/chapter12/ng-run-outside-angular` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-run-outside-angular
```

This should open the app in a new browser tab, and you should see the following:

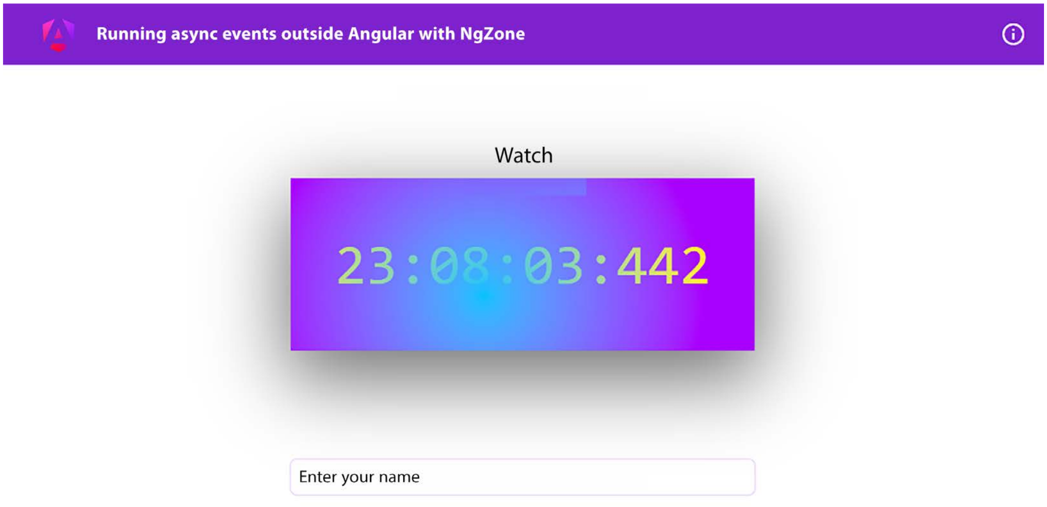


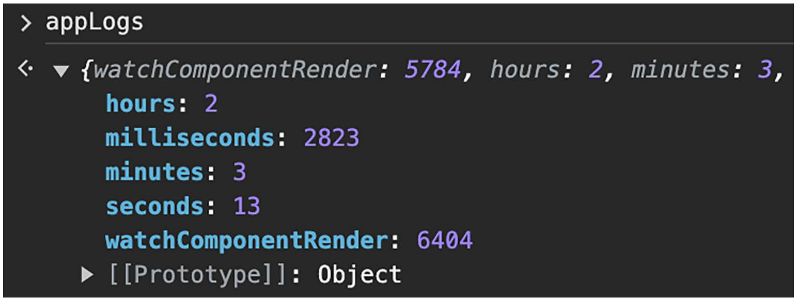
Figure 12.13: The ng-run-outside-angular app running on http://localhost:4200

Now that we have the app running, let's see the steps of the recipe in the next section.

How to do it...

We have an app that shows a watch. However, the change detection right now in the app is not optimal, and we have plenty of room for improvement. We'll try to remove any unnecessary change detection using the `runOutsideAngular` method from `ngZone`. Let us get started:

1. The clock values are constantly updating. Thus, we have change detection running for each update cycle. Open the Chrome DevTools and switch to the **Console** tab. Type **appLogs** and press *Enter* to see how many times the change detection has run for `WatchComponent` and for the components rendering hours, minutes, seconds, and milliseconds. It should look like this:



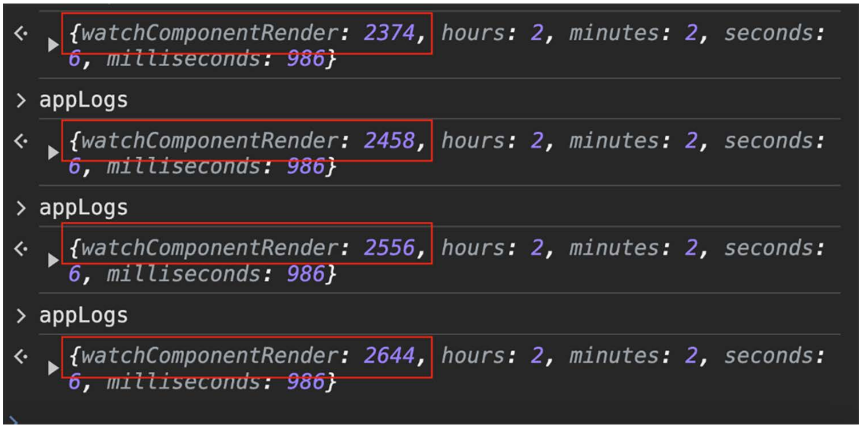
```
> appLogs
<· ▼ {watchComponentRender: 5784, hours: 2, minutes: 3,
      hours: 2
      milliseconds: 2823
      minutes: 3
      seconds: 13
      watchComponentRender: 6404
    ▶ [[Prototype]]: Object
```

Figure 12.14: The appLogs object reflecting the number of change detection runs

2. To measure performance, let's reduce our observation span in terms of time. Let us add some code to turn off the interval timer in 4 seconds from the app's start for the clock. Modify the `watch-box.component.ts` file, as follows:

```
...
@Component({...})
export class WatchBoxComponent implements OnInit {
  ...
  ngOnInit(): void {
    this.intervalTimer = setInterval(() => {
      this.timer();
  }, 1);
    setTimeout(() => {
      clearInterval(this.intervalTimer);
    }, 4000);
  }
  ...
}
```

3. Refresh the app and wait for 4 seconds for the clock to stop. Then, type `appLogs` multiple times in the **Console** tab, press *Enter*, and see the results. The clock stops but the animation is still running. You should see that change detection for the **watchComponentRender** key still increases, as follows:



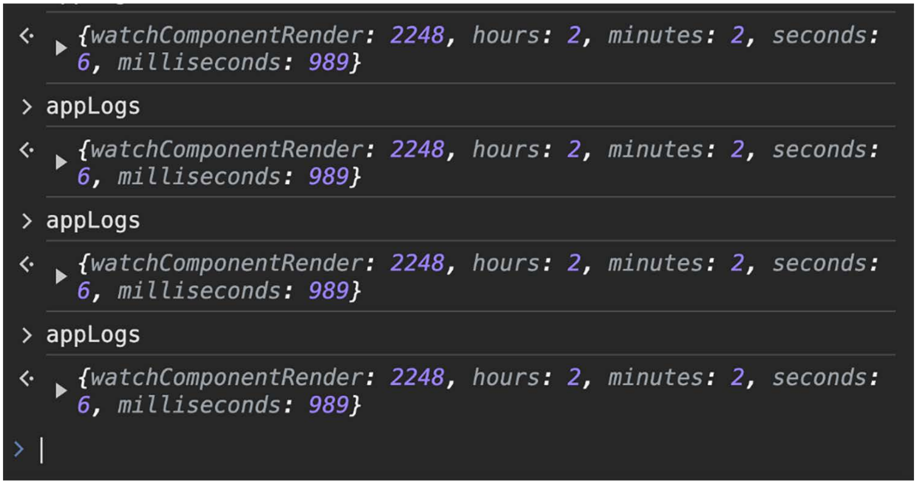
```
<· ▶ {watchComponentRender: 2374, hours: 2, minutes: 2, seconds: 6, milliseconds: 986}
> appLogs
<· ▶ {watchComponentRender: 2458, hours: 2, minutes: 2, seconds: 6, milliseconds: 986}
> appLogs
<· ▶ {watchComponentRender: 2556, hours: 2, minutes: 2, seconds: 6, milliseconds: 986}
> appLogs
<· ▶ {watchComponentRender: 2644, hours: 2, minutes: 2, seconds: 6, milliseconds: 986}
```

Figure 12.15: Change detection still running for the watch component

4. Let's also stop the animation inside the watch after 4 seconds. Update the `watch.component.ts` file, as follows:

```
...
export class WatchComponent implements OnInit {
  ...
  ngOnInit(): void {
    this.intervalTimer = setInterval(...}, 30);
    setTimeout(() => {
      clearInterval(this.intervalTimer);
    }, 4000);
  }
  ...
}
```

Refresh the app and wait for the animation to stop. Have a look at the `appLogs` object in the Chrome DevTools. You should see that change detection stops for the `watch` key, as follows:



```
<· ▶ {watchComponentRender: 2248, hours: 2, minutes: 2, seconds:
     6, milliseconds: 989}
>  appLogs
<· ▶ {watchComponentRender: 2248, hours: 2, minutes: 2, seconds:
     6, milliseconds: 989}
>  appLogs
<· ▶ {watchComponentRender: 2248, hours: 2, minutes: 2, seconds:
     6, milliseconds: 989}
>  appLogs
<· ▶ {watchComponentRender: 2248, hours: 2, minutes: 2, seconds:
     6, milliseconds: 989}
>
```

Figure 12.16: Change detection stops after we stop the animation interval

5. The desired outcome is to keep running the animation and the clock and not have additional change detection cycles running. For this, let's just stop the watch for now. To do that, update the `watch-box.component.ts` file, as follows:

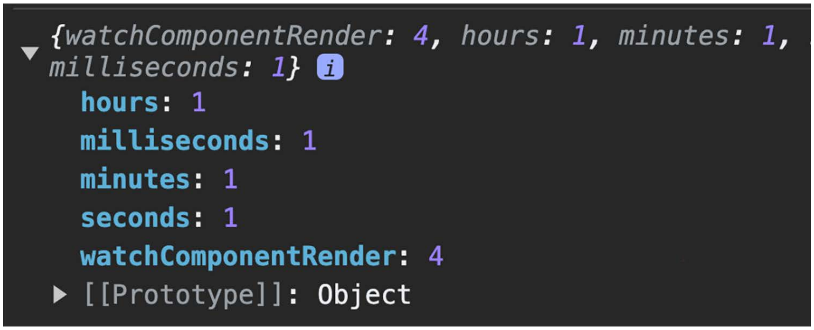
```
...
@Component({...})
export class WatchBoxComponent implements OnInit {
  ...
  ngOnInit(): void {
    // this.intervalTimer = setInterval(() => {
    //   this.timer();
    // }, 1);
    // setTimeout(() => {
    //   clearInterval(this.intervalTimer);
    // }, 4000);
  }
}
```

Since we have now stopped the clock, the values for `appLogs` for the `watchComponentRender` key are now only based on the animation for these 4 seconds. Refresh the app and wait for the animation to stop. Type `appLogs` in the Chrome DevTools (in the **Console** tab). You should now see a value between `250` and `270` for the `watchComponentRender` key.

6. Let us avoid running change detection on the animation by running the interval outside the `ngZone` service. We will use the `runOutsideAngular` method for this. Update the `watch.component.ts` file, as follows:

```
import {
  ...
  ViewChild,
  inject,
  NgZone,
} from '@angular/core';
@Component({...})
export class WatchComponent implements OnInit {zone = inject(NgZone);
  ...
  ngOnInit(): void {
  if (!window['appLogs']) {
    window['appLogs'] = {};
  }
  window['appLogs'][ watchComponentRender] = 0;
    this.zone.runOutsideAngular(() => {
   ...
      setTimeout(() => {
        clearInterval(this.intervalTimer);
      }, 4000);
    });
  }
  ...
}
```

Refresh the app and wait for about 5 seconds. If you check the `appLogs` object now, you should see a decrease in the overall number of change detection runs for each of the properties, as follows:



```
▼ {watchComponentRender: 4, hours: 1, minutes: 1,
  milliseconds: 1}
    hours: 1
    milliseconds: 1
    minutes: 1
    seconds: 1
    watchComponentRender: 4
  ▶ [[Prototype]]: Object
```

Figure 12.17: The appLogs object after using runOutsideAngular() in WatchComponent

Yay! Notice that the value for the `watch` key in the `appLogs` object has decreased from about `260` to `4` now. This means that our animation now does not contribute to change detection at all, and the `watch` component only renders 4 times in 4 seconds.

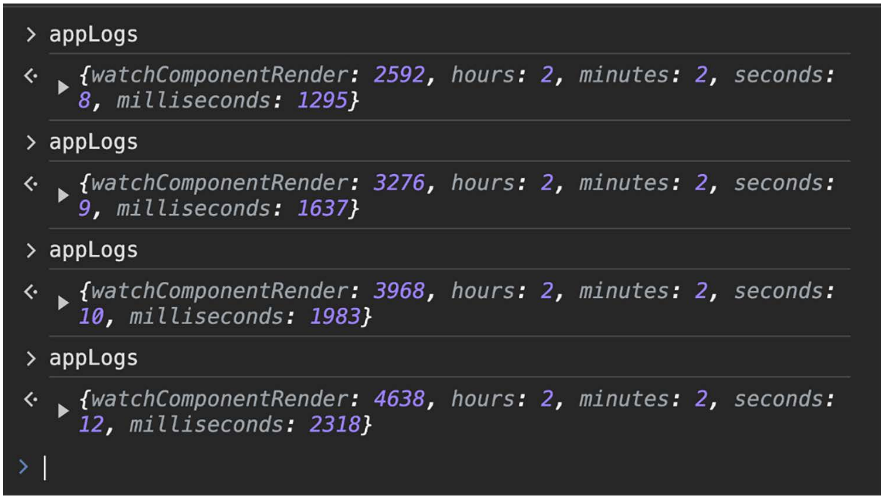
7. Remove the usage of `clearInterval` from the animation for the `WatchComponent` class. Thus, the background circle (blue circle) animation should start again. Modify the `watch.component.ts` file, as follows:

```
...
@Component({...})
export class WatchComponent implements OnInit {
  ...
  ngOnInit(): void {
    ...
    this.ngZone.runOutsideAngular(() => {
      this.intervalTimer = setInterval(() => {
        this.animate();
      }, 30);
      setTimeout(() => { <-- remove this
        clearInterval(this.intervalTimer);
      }, 4000);
    });
  }
  ...
}
```

8. Finally, remove the usage of `clearInterval` from the `WatchBoxComponent` class and uncomment `setInterval` to run the clock. Update the `watch-box.component.ts` file, as follows:

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-watch-box',
  templateUrl: './watch-box.component.html',
  styleUrls: ['./watch-box.component.scss'],
})
export class WatchBoxComponent implements OnInit {
  name = '';
  time = {
    hours: 0,
    minutes: 0,
    seconds: 0,
    milliseconds: 0,
  };
  intervalTimer;
  constructor() {}
  ngOnInit(): void {
    this.intervalTimer = setInterval(() => {
      this.timer();
    }, 1);
    setTimeout(() => { //<-- Remove this
      clearInterval(this.intervalTimer);
    }, 4000);
  }
  ...
}
```

Refresh the app and check the value of the `appLogs` object after a few seconds, multiple times. You should see something like this:



```
> appLogs
<· ▶ {watchComponentRender: 2592, hours: 2, minutes: 2, seconds:
     8, milliseconds: 1295}
> appLogs
<· ▶ {watchComponentRender: 3276, hours: 2, minutes: 2, seconds:
     9, milliseconds: 1637}
> appLogs
<· ▶ {watchComponentRender: 3968, hours: 2, minutes: 2, seconds:
     10, milliseconds: 1983}
> appLogs
<· ▶ {watchComponentRender: 4638, hours: 2, minutes: 2, seconds:
     12, milliseconds: 2318}
> |
```

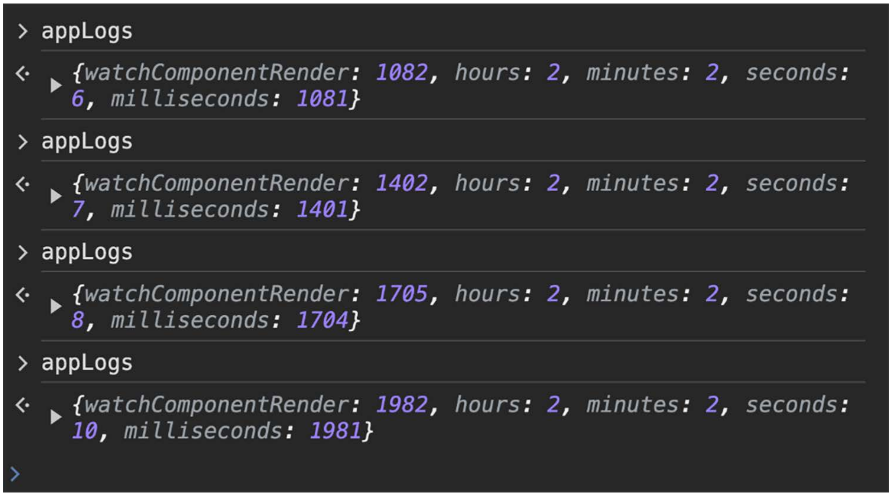
Figure 12.18: The appLogs object after performance optimization with runOutsideAngular()

Looking at the preceding screenshot, you might ask, "*Ahsan! What is this? We still have an enormous number for the change detection runs for the* `watchComponentRender` *key compared to the milliseconds passed. How is this performant exactly?*" Glad you asked! I will tell you why in the *How it works...* section.

9. As a last step, stop the Angular server and run the following command to start the server in production mode:

```
npm run serve:prod ng-run-outside-angular
```

10. Navigate to `https://localhost:4200` again. Wait for a few seconds and then check the `appLogs` object in the **Console** tab multiple times. You should see the object, as follows:



```
> appLogs
<· ▶ {watchComponentRender: 1082, hours: 2, minutes: 2, seconds:
     6, milliseconds: 1081}
> appLogs
<· ▶ {watchComponentRender: 1402, hours: 2, minutes: 2, seconds:
     7, milliseconds: 1401}
> appLogs
<· ▶ {watchComponentRender: 1705, hours: 2, minutes: 2, seconds:
     8, milliseconds: 1704}
> appLogs
<· ▶ {watchComponentRender: 1982, hours: 2, minutes: 2, seconds:
     10, milliseconds: 1981}
>
```

Figure 12.19: The appLogs object using the production build

Boom! If you look at the preceding screenshot, you should see that the change detection run count for the `watchComponentRender` key is always just a few cycles more than the `milliseconds` key. This means that the `WatchComponent` class is (almost) only re-rendered whenever we have the value of the `@Input() milliseconds` binding updated. But why a few cycles more? See the next section to understand how it works!

How it works...

In this recipe, we begin by looking at the `appLogs` object, which contains some key-value pairs. The value for each key-value pair represents the number of times Angular ran change detection for a particular component. The `hours`, `milliseconds`, `minutes`, and `seconds` keys represent the `WatchTimeComponent` instance for each of the values shown on the clock. The `watchComponentRender` key represents the `WatchComponent` instance's change detection cycles.

At the beginning of the recipe, we see that the value of the `watch` key is more than twice the value of the `milliseconds` key. Why do we care about the `milliseconds` key at all? Because the `@Input()` attribute binding `milliseconds` changes most frequently in our application—that is, it changes every 1 **Millisecond** (**ms**). The second most frequently changed values are the `xCoordinate` and `yCoordinate` properties within the `WatchComponent` class, which change every 30 ms. The `xCoordinate` and `yCoordinate` values are not bound directly to the template (the HTML) because they change the **CSS** variables of the `stopWatch` view child. This happens inside the `animate` method, as follows:

```
el.style.setProperty('--x', `${this.xCoordinate}px`);
el.style.setProperty('--y', `${this.yCoordinate}px`);
```

Thus, changing these values should not trigger change detection. We begin by limiting the testing time to run the clock, using the `clearInterval` method in the `WatchBoxComponent` class so that the clock stops within 4 seconds, and we can evaluate the numbers. In *Figure 12.15*, we see that even after the clock stops, the change detection mechanism keeps triggering for the `WatchComponent` class. This increases the count for the `watch` key in the `appLogs` object as time passes. We then stop the animation by using `clearInterval` in the `WatchComponent` class. This stops the background (blue circle) animation after 4 seconds as well. In *Figure 12.16*, we see that the count for the `watch` key stops increasing after the animation stops.

We then try to see the count of change detection only based on the animation. In *step 6*, we stop the clock. Therefore, we only get a count based on the animation in the `appLogs` object for the `watch` key, which is a value between `250` and `270`.

We then introduce the magic `runOutsideAngular` method into our code. This method is part of the `NgZone` service. The `NgZone` service is packaged with the `@angular/core` package. The `runOutsideAngular` method accepts a method as a parameter. This method is executed outside the Angular zone. This means that the `setTimeout` and `setInterval` methods used inside the `runOutsideAngular` method do not trigger the Angular change detection cycle. But technically, why are we running this `setInterval` outside the zone? Because our interval calls the `animate` method, which updates the CSS variables `--x` and `--y`. And since they automatically trigger the animation and no other property of the `WatchComponent` class must be shown in the UI (requiring a re-render), we can move this code to the `runOutsideAngular` method. You can see in *Figure 12.17* that the count drops to 4 after using the `runOutsideAngular` method.

We then remove the `clearInterval` usage from both the `WatchBoxComponent` and the `WatchComponent` classes—that is, to run the clock and the background (blue circle) animation again, as we did in the beginning. In *Figure 12.18*, we see that the count for the `watch` key is around twice the value of the `milliseconds` key. Now, why exactly is it that it's around double? This is because, in development mode, Angular runs the change detection mechanism *twice* to ensure there are no *side effects*. For example, an update to a `state` property initiating a change detection in another (child) component, and so on. Therefore, in *step 9* and *step 10*, we run the application in production mode, and in *Figure 12.19*, we see that the value for the `watch` key is just a few cycles greater than the value for the `milliseconds` key, which means that the animation does not trigger any change detection for our application anymore.

But why does the `watchComponentRender` key have a few more cycles compared to the `milliseconds` key? That is because the `WatchComponent` is the parent component of the component displaying milliseconds (`WatchTimeComponent`). There could be a few change detection cycles based on browser interactions, but if you refresh the app and have no interaction at all with the app, the difference between `watchComponentRender` to the milliseconds count would be as low as one change detection cycle in production mode.

Brilliant, isn't it? If you find this recipe useful, do let me know on my socials.

Now that you understand how it works, see the next section for further reading.

See also

- The `NgZone` official documentation: `https://angular.io/api/core/NgZone`
- Angular `ChangeDetectorRef` official documentation: `https://angular.io/api/core/ChangeDetectorRef`

Using trackBy for lists with *ngFor

Lists are an essential part of most of the apps we build today. If you're building an Angular app, there's a great chance you will use the `*ngFor` directive at some point for rendering lists. The `*ngFor` directive allows us to loop over arrays or objects generating HTML for each item. However, if we are rendering large lists, using `*ngFor` without caution may cause performance issues, especially when the source for `*ngFor` is changed completely (the entire array is replaced). In this recipe, we'll learn how we can improve the performance of lists using the `*ngFor` directive with the `trackBy` function. Let's get started.

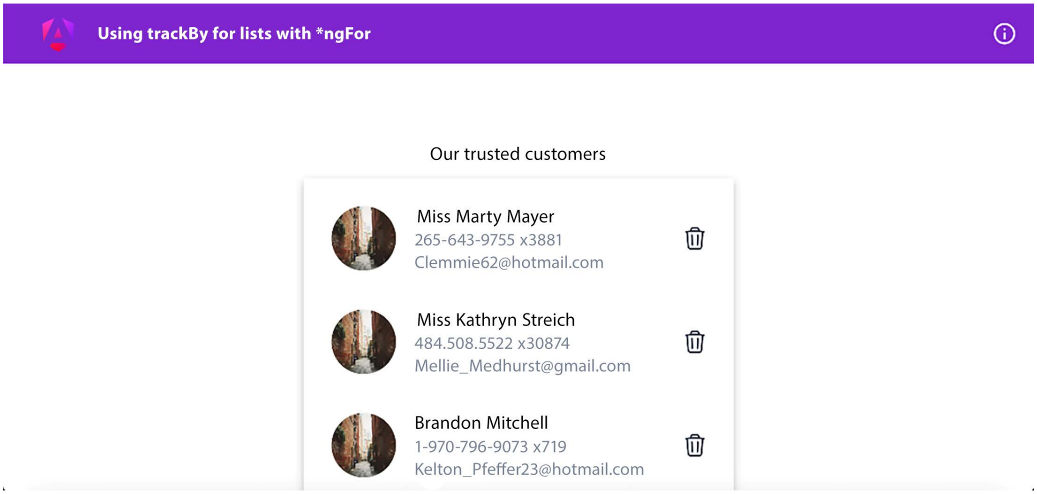
Getting ready

The app that we are going to work with resides in `start/apps/chapter12/ng-for-trackby` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-for-trackby
```

This should open the app in a new browser tab, and you should see the following:



Using trackBy for lists with *ngFor

Our trusted customers

Miss Marty Mayer
265-643-9755 x3881
Clemmie62@hotmail.com

Miss Kathryn Streich
484.508.5522 x30874
Mellie_Medhurst@gmail.com

Brandon Mitchell
1-970-796-9073 x719
Kelton_Pfeffer23@hotmail.com

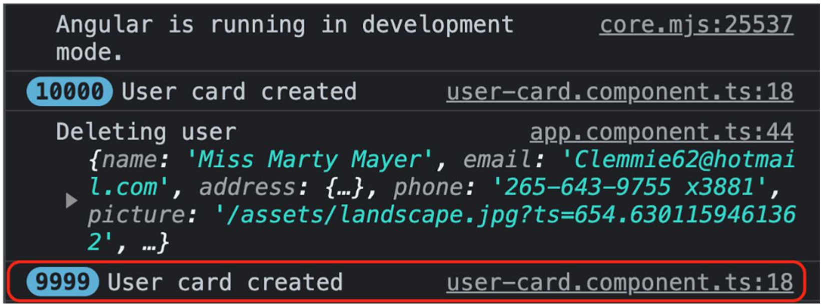
Figure 12.20: The ng-for-trackby app running on http://localhost:4200

Now that we have the app running, let's see the steps of the recipe in the next section.

How to do it...

We have an app that has a list of 10,000 users displayed on the view. Since we're not using a virtual scroll (from `@angular/material`, for example), and are using a standard `*ngFor` list, we do face some performance issues now. Notice that when you refresh the app, even after the loader is hidden, you see a blank white box for about 2–3 seconds before the list appears. If your machine is stuck for too long, you can modify the value of the `USERS_LIMIT` variable in the `data.service.ts` file. Let us start the recipe to reproduce the performance issues, and we will fix them afterward:

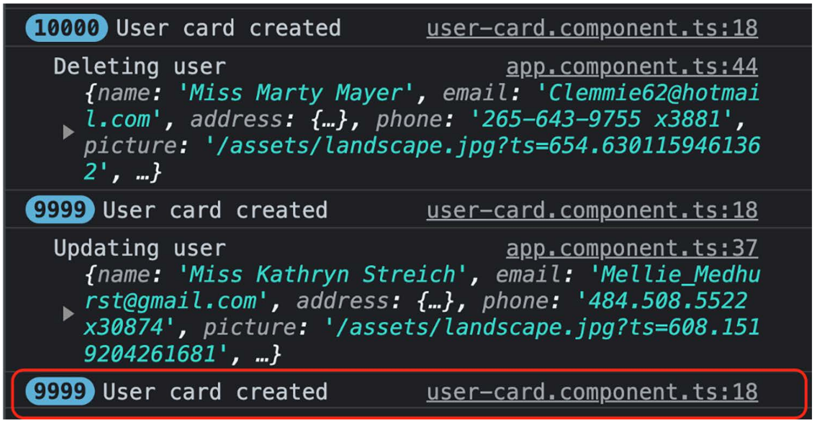
1. First, open the Chrome DevTools and look at the **Console** tab. You should see the **User card created** message logged 10,000 times. This message will be logged any time a user card component (instance of the `UserCardComponent` class) is created/initiated.
2. Now, delete the first item by using the *delete* button on the card. You should see the same message (**User card created**) logged again 9,999 times now, as shown in the following screenshot. This means we recreate the `list-item` component for the remaining 9,999 items:



```
Angular is running in development        core.mjs:25537
mode.
10000 User card created          user-card.component.ts:18
Deleting user                            app.component.ts:44
  {name: 'Miss Marty Mayer', email: 'Clemmie62@hotmai
▶ l.com', address: {…}, phone: '265-643-9755 x3881',
  picture: '/assets/landscape.jpg?ts=654.630115946136
  2', …}
9999 User card created           user-card.component.ts:18
```

Figure 12.21: Logs shown again after deleting an item

3. Now, click the first item (this updates the first item according to the existing code). You should again see the **User card created** logs, as shown in *Figure 12.22*. This means we recreate all the 9,999 list items when updating any item from the list. You will notice that the update to the first item's name in the **User Interface** (**UI**) is reflected in about 2–3 seconds:



```
10000 User card created          user-card.component.ts:18
  Deleting user                        app.component.ts:44
    {name: 'Miss Marty Mayer', email: 'Clemmie62@hotmai
  ▶ l.com', address: {…}, phone: '265-643-9755 x3881',
    picture: '/assets/landscape.jpg?ts=654.630115946136
    2', …}
9999 User card created           user-card.component.ts:18
  Updating user                        app.component.ts:37
    {name: 'Miss Kathryn Streich', email: 'Mellie_Medhu
  ▶ rst@gmail.com', address: {…}, phone: '484.508.5522
    x30874', picture: '/assets/landscape.jpg?ts=608.151
    9204261681', …}
9999 User card created           user-card.component.ts:18
```

Figure 12.22: Logs shown again after updating an item

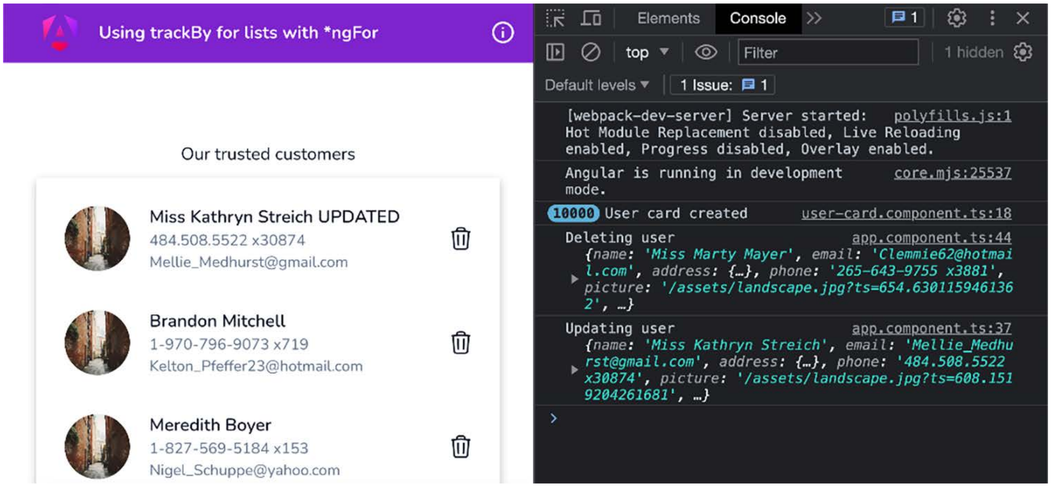
4. Now, let's fix the performance issue by using the `trackBy` function. Open the `users-list.component.ts` file and update it, as follows:

```
...
export class UsersListComponent {
  @Input() listItems: AppUserCard[] = [];
  @Output() itemClicked = new EventEmitter<AppUserCard>();
  @Output() itemDeleted = new EventEmitter<AppUserCard>();
  trackByFn(_index: number, item: AppUserCard) {
    return item.id;
  }
}
```

5. Now, update the `users-list.component.html` file to use the `trackByFn` method we just created, as follows:

```
<h4 class="heading">Our trusted customers</h4>
<ul class="list list-group p-2">
  <li class="list__item list-group-item" *ngFor="let item of
    listItems; trackBy: trackByFn">
    ...
  </li>
</ul>
```

6. Now, refresh the app, delete the first item, and then click the (new) first list item to update it. You will notice that the item is updated immediately, and we do not log the **User card created** message again anymore, as shown in *Figure 12.23*:



Using trackBy for lists with *ngFor

Our trusted customers

Miss Kathryn Streich UPDATED
484.508.5522 x30874
Mellie_Medhurst@gmail.com

Brandon Mitchell
1-970-796-9073 x719
Kelton_Pfeffer23@hotmail.com

Meredith Boyer
1-827-569-5184 x153
Nigel_Schuppe@yahoo.com

```
[webpack-dev-server] Server started:   polyfills.js:1
Hot Module Replacement disabled, Live Reloading
enabled, Progress disabled, Overlay enabled.
Angular is running in development     core.mjs:25537
mode.
10000 User card created        user-card.component.ts:18
Deleting user                        app.component.ts:44
  {name: 'Miss Marty Mayer', email: 'Clemmie62@hotmai
▸ l.com', address: {…}, phone: '265-643-9755 x3881',
  picture: '/assets/landscape.jpg?ts=654.630115946136
  2', …}
Updating user                        app.component.ts:37
  {name: 'Miss Kathryn Streich', email: 'Mellie_Medhu
▸ rst@gmail.com', address: {…}, phone: '484.508.5522
  x30874', picture: '/assets/landscape.jpg?ts=608.151
  9204261681', …}
>
```

Figure 12.23: No further logs after updating an item using the trackBy function

Great! You now know how to use the `trackBy` function with the `*ngFor` directive to optimize the performance of lists in Angular. To understand all the magic behind the recipe, see the next section.

How it works...

The `*ngFor` directive allows us to loop over **iterables** and render multiple HTML elements or components. When working with an array of primitives (Boolean, string, or number values), Angular tracks each item by its value to identify elements. However, when dealing with objects, Angular tracks them by memory location, like how JavaScript handles object equality checks. This means that if you just change a property in an object of the array, it will not re-render the template for that object. However, if you provide a new object in its place (different reference in memory), the content for the item will be re-rendered. This is what we do in this recipe to reproduce the performance issue. Since we replace the entire array when we update or delete an item, Angular treats the array as a new resource to iterate over. In the `data.service.ts` file, we have the following code for the `updateUser` method inside the service named `DataService`:

```
updateUser(updatedUser: AppUserCard) {
  this.users = this.users.map((user) => {
    if (user.id === updatedUser.id) {
      return {
        ...updatedUser,
      };
    }
    return { ...user };
  });
}
```

Notice that we use the object spread operator (`{ … }`) to return a new object for each item in the array. This ends up telling the `*ngFor` directive to re-render the UI for each item in the `listItems` array in the `UserListComponent` class. Suppose you are already rendering 1,000 users. If you search for a term that returns 100 users from the server, ideally, Angular should not re-render those 100 users as they were already rendered in the view. Angular, however, would re-render the UI for all the list items because of the following potential reasons (but not limited to these):

- The sorting/placement of the users could have changed.
- The length of the users could have changed.

Now, we want to avoid using the object reference as the unique identifier for each list item. For our use case, we know that each user's ID is unique; therefore, we use the `trackBy` function to tell Angular to use the user's ID as the unique identifier. Now, even if we return a new object for each user after a user update from the `updateUser` method (as previously shown), Angular does not re-render all the list items. This is because the new objects (users) have the same ID and Angular uses it to track them. Pretty cool, right?

Now that you have learned how the recipe works, see the next section to view a link for further reading.

See also

- The `NgForOf` official documentation: `https://angular.io/api/common/NgForOf`

Moving heavy computation to pure pipes

In Angular, we have a particular way of writing components. Since Angular is heavily opinionated, we already have a lot of guidelines from the community and the Angular team on what to consider when writing components—for example, making HTTP calls directly from a component is considered a *not-so-good* practice. Similarly, if we have heavy computation in a component that triggers with each change detection cycle, this is also not considered a good practice. Imagine that the view depends upon a transformed version of the data using a computation constantly. This would cause a lot of computation and processing for each render cycle. One good technique can be to move the heavy computation using Angular (pure) pipes (especially if the computation happens with each change detection cycle).

Getting ready

The app that we are going to work with resides in `start/apps/chapter12/ng-pipes-perf` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-pipes-perf
```

This should open the app in a new browser tab, and you should see the following:

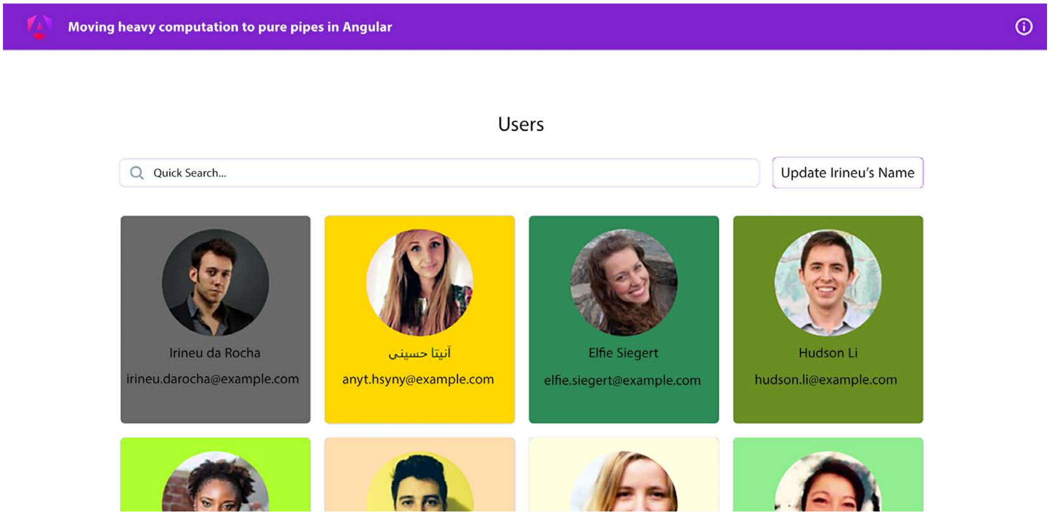

Moving heavy computation to pure pipes in Angular
Users
Quick Search...
Update Irineu's Name
Irineu da Rocha
irineu.darocha@example.com
آنیتا حسینی
anyt.hsyny@example.com
Elfie Siegert
elfie.siegert@example.com
Hudson Li
hudson.li@example.com

Figure 12.24: The ng-pipes-perf app running at http://localhost:4200

Click the button that says **Click Me** Or try to search for some user. You'll see that the app is too slow and gets hung often. Now that we have the project served on the browser, let's see the steps of the recipe in the next section.

How to do it...

The app we're working with has some performance issues, particularly with the `UserCardComponent` class. This is because it uses a getter function, `randomColor`, to generate a random color for its background. Behind the scenes, that function uses a `factorial` function to add more processing time. But that is just to demonstrate a component that can cause the UI to hang if there is some complex calculation happening along with multiple change detections being triggered. We're going to add some code to monitor how many times the `randomColor` getter gets called. This will show the number of times the change detection in Angular gets triggered by default. We will also fix the issue and make it more performant (as much as we can) by detaching the change detector completely from a particular component. Let's get started:

1. First, let's make sure that the app is just slow enough for your machine that it makes your laptop/PC hang. Open the `src/app/app.config.ts` file and adjust the value of the `RANDOMIZATION_COUNT` token from `9` to whatever suits you best.
2. Then, try to search for a user named `Irineu` by entering their name in the search box. You'll notice that the app still hangs and that it takes a few seconds to show the user. You'll also notice that you don't even see the letters in the search box as you type them. That is, there is a delay in rendering.

   Let's add some logic to the code. We'll check how many times Angular calls the `idUsingFactorial` method when the page loads.

3. Let's create a service that we will use to track the number of times a particular user card for a particular person is called. Run the following command from the workspace's root folder to create a service:

   ```
   cd start && nx g s services/logs --project ng-pipes-perf
   ```

   Choose `@schematics/angular:service` when asked.

4. Update the content of the `src/app/services/logs.service.ts` file as follows:

   ```
   import { Injectable } from '@angular/core';

   @Injectable({
     providedIn: 'root'
   })
   export class LogsService {
     logs: Record<string, number> = {}

     updateLogEntry(email: string) {
       if (this.logs[email] === undefined) {
         this.logs = {
           ...this.logs,
           [email]: 1
         }
       } else {
         this.logs = {
           ...this.logs,
           [email]: this.logs[email] + 1
         }
       }
     }
   }
   ```

5. Now, inject the `LogService` in the `src/app/component/user-card/user-card.component.ts` file. We will also create a getter (`log`) function to get the count for the user, and we will update the count whenever the `randomColor` getter is called. Update the mentioned file as follows:

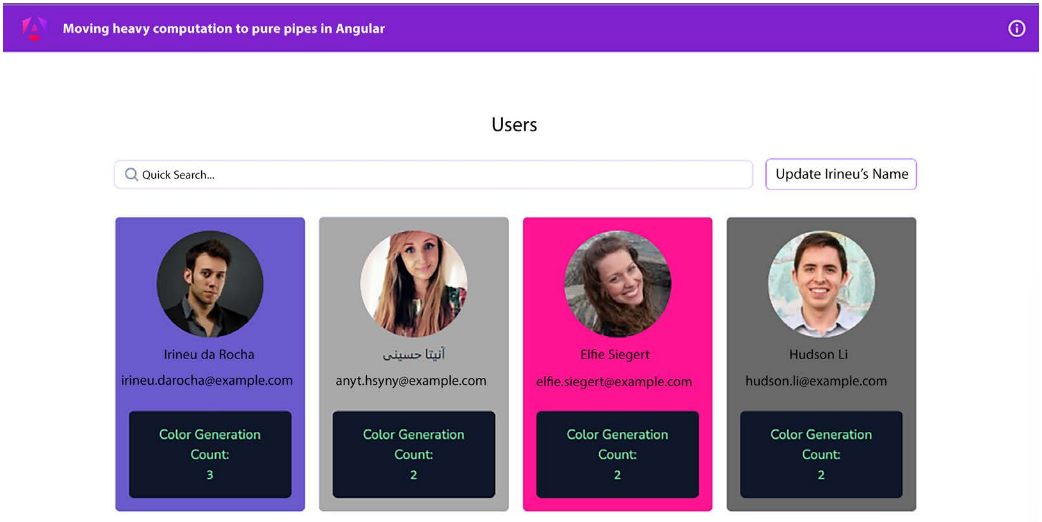
   ```
   ...
   import { LogsService } from '../../services/logs.service';

   @Component({...})
   export class UserCardComponent {
     ...
     randomizationCount = inject(RANDOMIZATION_COUNT);
   ```

```
  logsService = inject(LogsService);

  get log() {
    return this.logsService.logs[this.user.email] ?? 0;
  }

  get randomColor() {
    this.logsService.updateLogEntry(this.user.email);
    ...
  }
}
```

6. Now, we will use the log in the template of the user card component to show the count. Update the `src/app/component/user-card/user-card.component.html` file as follows:

```
<div [style.backgroundColor]="randomColor"...>
  <img ...>
  <div class="card-body flex-1">...</div>
  <div class="p-4 bg-slate-900 text-green-300 rounded-md h-fit">
    <div>
      Color Generation Count:
    </div>
    <pre>{{log}}</pre>
  </div>
</div>
```

If you look at the application now, you should see the **Color Generation Count** as follows:


Moving heavy computation to pure pipes in Angular
Users
Quick Search...
Update Irineu's Name

Irineu da Rocha
irineu.darocha@example.com
Color Generation Count:
3

آنیتا حسینی
anyt.hsyny@example.com
Color Generation Count:
2

Elfie Siegert
elfie.siegert@example.com
Color Generation Count:
2

Hudson Li
hudson.li@example.com
Color Generation Count:
2

Figure 12.25: Color Generation Count being shown on page load

7. Now, click the **Update Irineu's Name** button. Then focus (click) the **Quick Search** input and then click outside. Repeat this a few times and you should see that the colors are regenerated, even though the cards are not supposed to be re-rendered. *Figure 12.26* shows how it should look:

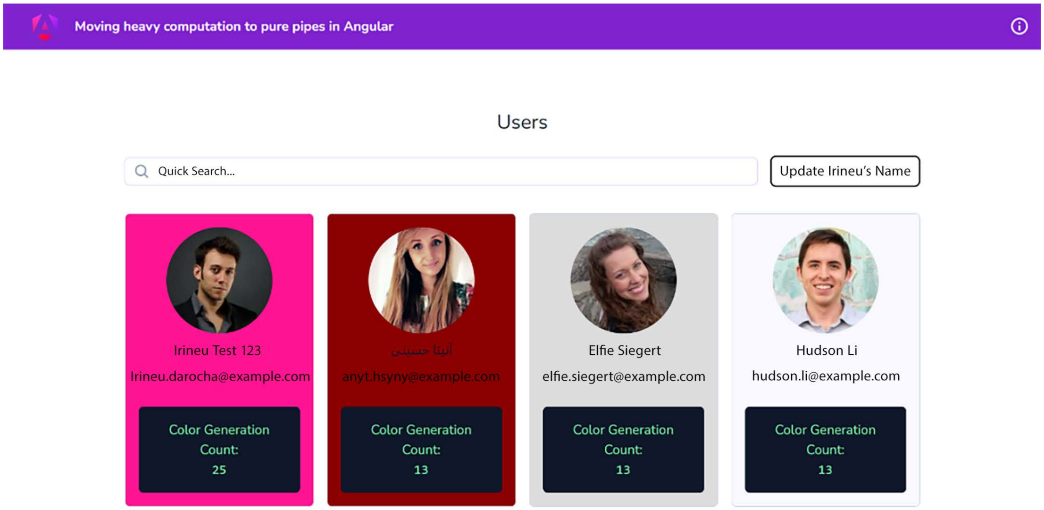

Moving heavy computation to pure pipes in Angular
Users
Quick Search...
Update Irineu's Name
Irineu Test 123
Irineu.darocha@example.com
Color Generation Count:
25
أنيتا حسيني
anyt.hsyny@example.com
Color Generation Count:
13
Elfie Siegert
elfie.siegert@example.com
Color Generation Count:
13
Hudson Li
hudson.li@example.com
Color Generation Count:
13

Figure 12.26: Logs after interacting with the app without searching for anything

Notice that if you start searching for something, you'll get even more re-renders. That's because each key up and/or key down event will trigger more re-renders.

8. To fix the issue, we will create an Angular pipe. We'll move the computation to generate the random colors for this user card to this Angular pipe. In the project root, run the following command in the terminal:

```
cd start && nx g pipe random-color --directory apps/chapter12/ng-pipes-perf/src/app/pipes
```

Use the `@schematics/angular:pipe` schematic when asked.

9. Now, move the `randomColor` getter function and the `factorial` function from the `user-card.component.ts` file to the Angular pipe's file, `pipes/random-color.pipe.ts`, as follows:

```
import { Pipe, PipeTransform, inject } from '@angular/core';
import { LogsService } from '../services/logs.service';
import { randColor } from '@ngneat/falso';
import { IUser } from '../interfaces/user.interface';
...
export class RandomColorPipe implements PipeTransform {
  logsService = inject(LogsService);
  factorial(n: number): number {
    if (n == 0 || n == 1) {
      return 1;
    } else {
      return n * this.factorial(n - 1);
    }
  }

  randomColor(email: string, randomizationCount: number) {
    this.logsService.updateLogEntry(email);
    let color;
    for (let i = 0; i < this.factorial(randomizationCount); i++) {
      color = randColor();
    }
    return color;
  }

  transform(randomizationCount: number, user: IUser):
    string | undefined {
    return this.randomColor(user.email, randomizationCount);
  }
}
```

10. Make sure to delete those functions (the `randomColor` getter and `factorial`) from the `src/app/component/user-card/user-card.component.ts` file. Remove any unused imports as well.
11. Now, add the `randomColor` pipe to the user card component in the `user-card.component.ts` file as follows:

```
...
import { RandomColorPipe } from '../../pipes/random-color.pipe';

@Component({
  selector: 'app-user-card',
  standalone: true,
  imports: [CommonModule, RouterModule, RandomColorPipe],
  ...
})
```

12. Now, update the `user-card.component.html` file to use the `randomColor` pipe instead of the getter we were using before. The code should look like this:

```
<div [style.backgroundColor]="randomizationCount | randomColor : user"
class="card flex flex-col max-w-sm mx-auto h-full duration-200 cursor-
pointer hover:border-purple-500 hover:shadow-md p-4 border border-
slate-300 rounded-md text-center" *ngIf="user" routerLink="/user/{{user.
uuid}}">
  ...
</div>
```

13. Now, refresh the app and repeat *step 7*. You will see that the color is only generated for the first card and the other cards do not re-render, as shown in *Figure 12.27*:

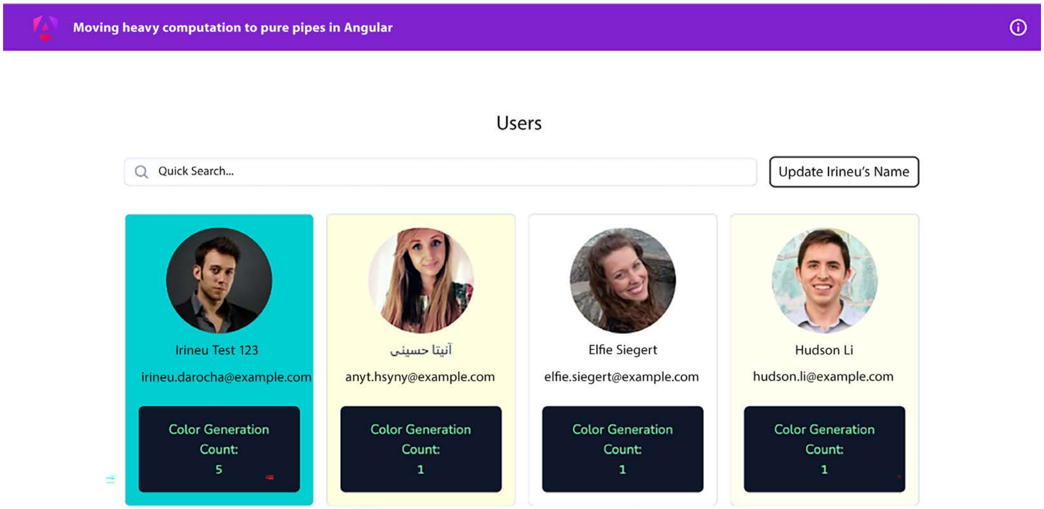

Moving heavy computation to pure pipes in Angular
Users
Quick Search...
Update Irineu's Name
Irineu Test 123
irineu.darocha@example.com
Color Generation Count: 5
آنيتا حسينى
anyt.hsyny@example.com
Color Generation Count: 1
Elfie Siegert
elfie.siegert@example.com
Color Generation Count: 1
Hudson Li
hudson.li@example.com
Color Generation Count: 1

Figure 12.27: Colors regenerating only for the first card

Boom! Now that you know how to optimize performance by moving heavy computation to pure Angular pipes, see the next section to understand how this works.

How it works…

As we know, Angular by default runs change detection on each browser event triggered in the app. And since we're using a `randomColor` getter in the component template (UI), this function runs each time Angular runs the change detection cycle. This causes more computation and performance issues. This would also hold true if we used a function call instead of a getter.

We can take a step back from the initial implementation and think about what the `randomColor` getter does. It works with a `for` loop based on the factorial of the `randomizationCount` property. This is just to add a lot of processing time for this example, but you can imagine any heavy computation involved in the change detection cycle causing the performance issue. And in these cases, we either use pure functions or perhaps memoization.

A pure function is a function that always returns the same output, given the same input. Memoization is a technique in which, if the inputs are not changed (that is, the function is called with the same inputs as the previous time), a cached output is returned, and the heavy computation is skipped. Luckily, Angular pure pipes are pure functions and memorized, as they are only called when the inputs change. If that's not the case, the pipe's transform function will not be called, and the component won't be re-rendered.

In the recipe, we move the computation to a newly created Angular pipe. The pipe's `transform` method receives `randomizationCount` as the first value, and the `user` (of type `IUser`) as the second input. The pipe then uses the `randomColor` method and, ultimately, the `factorial` method to calculate a random color. When we start typing in the search box, the values for the user cards do not change. This results in the pipe not being triggered until we get back a new set of users based on the search query. Once we get the result, the user cards are re-rendered and hence we get new colors for them. As a result, there is no unnecessary computation running due to browser events, thus optimizing performance and unblocking the UI thread.

See also

- Angular pure and impure pipes official documentation: `https://angular.io/guide/pipes#pure-and-impure-pipes`

Using web workers for heavy computation

If your Angular application does a lot of computation during an action, there's a high chance that it will block the UI thread. This will cause a lag in rendering the UI because it blocks the main JavaScript thread. Web workers allow us to run heavy computation in the background thread, thus freeing the UI thread as it is not blocked. In this recipe, we're going to use an application that does a heavy computation in the `UserService` class. It creates a unique ID for each user card and saves it into `localStorage`. However, it loops a couple of thousand times before doing so, which causes our application to hang for a while. In this recipe, we'll move the heavy computation from the components to a web worker and will also add a fallback in case web workers aren't available.

Getting ready

The app that we are going to work with resides in `start/apps/chapter12/ng-ww-perf` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to serve the project:

```
npm run serve ng-ww-perf
```

This should open the app in a new browser tab, and you should see the following:

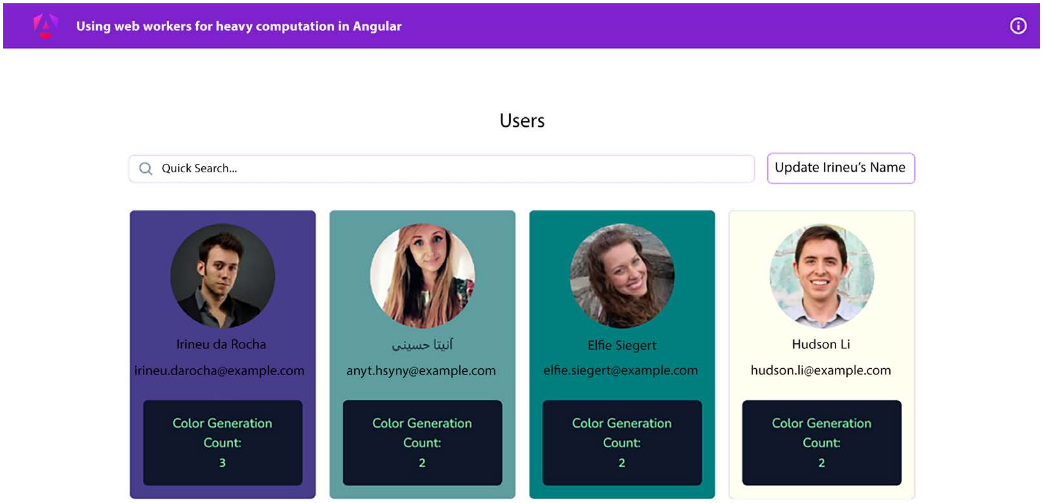

Using web workers for heavy computation in Angular
Users
Quick Search...
Update Irineu's Name
Irineu da Rocha
irineu.darocha@example.com
Color Generation Count: 3
أنيتا حسيني
anyt.hsyny@example.com
Color Generation Count: 2
Elfie Siegert
elfie.siegert@example.com
Color Generation Count: 2
Hudson Li
hudson.li@example.com
Color Generation Count: 2

Figure 12.28: The ng-ww-perf app running at http://localhost:4200

Now that we have the app running, let's see the steps of the recipe in the next section.

How to do it...

Once you open the app, you'll notice that it takes some time before the user cards are rendered. We can also see the number of times the random color for the user card background is generated. If you type something, or click the **Update Irineu's Name** button, you will see that the UI thread is blocked until we have some computation finished. The culprit is the `randomColor` getter method in the `UserCardComponent` class. This ultimately generates a random color after running a `for` loop based on the `RANDOMIZATION_COUNT` token's value before rendering the color. This happens inside the `generateRandomColor` method inside the `src/app/utils.ts` file. Let's start the recipe to improve the performance of the app. We'll start by implementing a web worker:

1. We'll first create a web worker. Run the following command in the workspace's root:

   ```
   cd start && nx generate web-worker workers/randomColor --project ng-ww-
   perf
   ```

   Choose `@nx/angular:web-worker` when asked.

2. Now, update the code inside the `workers/random-color.worker.ts` file as follows:

```
/// <reference lib="webworker" />

import { generateRandomColor } from "../utils";

type RandomColorIncomingEvent = {
  data: {
    randomizationCount: number
  }
}

export type RandomColorOutgoingEvent = { data: { color: string } };

addEventListener('message', ({ data }:
  RandomColorIncomingEvent) => {
  const {
    randomizationCount
  } = data;
  console.log('inside the worker', data)
  if (!randomizationCount) {
    return;
  }
  const color = generateRandomColor(randomizationCount);
  postMessage({
    color
  });
});

export const getRandomColorWorker = () => {
  if (typeof Worker !== undefined) {
    return new Worker(new URL('./random-color.worker', import.meta.url), {
      type: 'module'
    })
  }
  return null;
}
```

3. Let's replace the `randomColor` getter in the `UserCardComponent` class with a regular property. Update the `user-card.component.ts` file as follows:

```
...
export class UserCardComponent implements OnInit, OnChanges {
  ...
  randomizationCount = inject(RANDOMIZATION_COUNT);
  randomColor = '';
  ...
}
```

   Make sure to remove the `randomColor` getter function. Otherwise, TypeScript will throw errors since we can't have a property and a getter method having the same name.

4. Now, we'll use the worker in the `user-card.component.ts` file. Update it, as follows:

```
import { Component, Input, OnInit, inject } from '@angular/core';
...
import { RANDOMIZATION_COUNT } from '../../tokens';
import { RandomColorOutgoingEvent , getRandomColorWorker } from '../../
workers/random-color.worker';

@Component({...})
export class UserCardComponent implements OnInit {
  @Input() user!: IUser;
  @Input() index = 0;
  logsService = inject(LogsService);
  randomizationCount = inject(RANDOMIZATION_COUNT);
  randomColor = '';
  worker: Worker | null = getRandomColorWorker();

  ngOnInit(): void {
    if (!this.worker) {
      return;
    }
     this.worker.onmessage = ({ data: { color } }:
       RandomColorOutgoingEvent) => {
      console.log(
        `received color ${color} from worker for user ${this.user.email}`
      );
      this.logsService.updateLogEntry(this.user.email);
      this.randomColor = color;
    };
  }
 ...
}
```

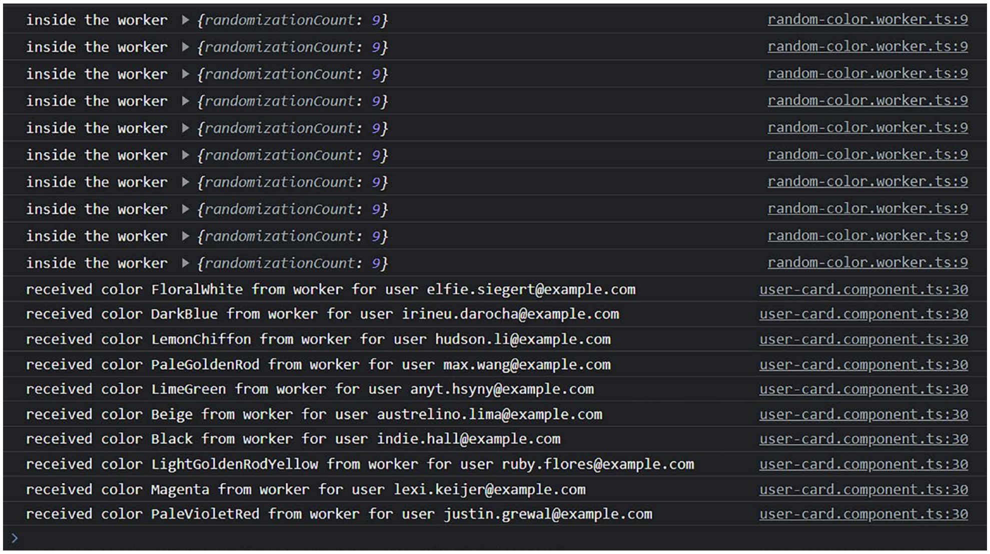
5. So far, we have just added a listener for when we'll receive the generated color from the worker. But first, we must send a message from the component to the worker, so it generates a random color. Update the `user-card.component.ts` file to use the `OnChanges` life cycle hook. We'll use it to send a message to the worker:

```
import { Component, Input, OnChanges, OnInit, SimpleChanges, inject }
from '@angular/core';
...
export class UserCardComponent implements OnInit, OnChanges {
  ...
  ngOnInit(): void {...}

  ngOnChanges(changes: SimpleChanges) {
    if (changes['user'].currentValue !==
      changes['user'].previousValue) {
      if (!this.worker) {
        this.randomColor = generateRandomColor(
          this.randomizationCount);
        return;
      }
      this.worker.postMessage({ randomizationCount:
        this.randomizationCount });
    }
  }
  ...
}
```

6. Finally, let's ensure that the worker is destroyed (terminated) when the respective user card is destroyed. Update the `user-card.component.ts` file as follows:

```
...
@ import { Component, Input, OnChanges, OnDestroy, OnInit, SimpleChanges,
inject } from '@angular/core';

export class UserCardComponent implements OnInit, OnChanges,
  OnDestroy {
  ...
  ngOnDestroy(): void {
    this.worker?.terminate();
  }

  get log() {...}
}
```

7. Refresh the app and notice how long it takes for the user cards to render. They should appear much faster than before. Also, you should be able to see the following logs reflecting the communication from the app to the web worker, and vice versa:



```
inside the worker ▶ {randomizationCount: 9}                              random-color.worker.ts:9
inside the worker ▶ {randomizationCount: 9}                              random-color.worker.ts:9
inside the worker ▶ {randomizationCount: 9}                              random-color.worker.ts:9
inside the worker ▶ {randomizationCount: 9}                              random-color.worker.ts:9
inside the worker ▶ {randomizationCount: 9}                              random-color.worker.ts:9
inside the worker ▶ {randomizationCount: 9}                              random-color.worker.ts:9
inside the worker ▶ {randomizationCount: 9}                              random-color.worker.ts:9
inside the worker ▶ {randomizationCount: 9}                              random-color.worker.ts:9
inside the worker ▶ {randomizationCount: 9}                              random-color.worker.ts:9
inside the worker ▶ {randomizationCount: 9}                              random-color.worker.ts:9
received color FloralWhite from worker for user elfie.siegert@example.com  user-card.component.ts:30
received color DarkBlue from worker for user irineu.darocha@example.com    user-card.component.ts:30
received color LemonChiffon from worker for user hudson.li@example.com     user-card.component.ts:30
received color PaleGoldenRod from worker for user max.wang@example.com     user-card.component.ts:30
received color LimeGreen from worker for user anyt.hsyny@example.com       user-card.component.ts:30
received color Beige from worker for user austrelino.lima@example.com      user-card.component.ts:30
received color Black from worker for user indie.hall@example.com           user-card.component.ts:30
received color LightGoldenRodYellow from worker for user ruby.flores@example.com  user-card.component.ts:30
received color Magenta from worker for user lexi.keijer@example.com        user-card.component.ts:30
received color PaleVioletRed from worker for user justin.grewal@example.com  user-card.component.ts:30
>
```

Figure 12.29: Logs showing messages to and from the app to web workers

Woohoo! The power of web workers! And now you know how to use web workers in an Angular app to move heavy computation to them. Since you've finished the recipe, see the next section on how this works.

How it works…

As we discussed in the recipe's description, web workers allow us to run and execute code in a separate thread from the main JavaScript (or UI thread). At the beginning of the recipe, whenever we refresh the app or search for a user, it blocks the UI thread. We sometimes see the loader being hung, or a white screen. This is until a random color is generated for each card. We begin the recipe by creating a web worker using the NX **Command-line Interface** (**CLI**). This creates a `random-color.worker.ts` file, which contains some boilerplate code to receive messages from the UI thread and send a message back to it as a response.

The CLI command also updates the `project.json` file by adding a `webWorkerTsConfig` property. The value against the `webWorkerTsConfig` property is the path to the `tsconfig.worker.json` file, and the CLI command also creates this `tsconfig.worker.json` file. If you open the `tsconfig.worker.json` file, you should see the following code:

```
/* To learn more about this file see: https://angular.io/config/tsconfig. */
{
  "extends": "../../../tsconfig.json",
  "compilerOptions": {
    "outDir": "../../../out-tsc/worker",
    "lib": [
      "es2018",
      "webworker"
    ],
    "types": []
  },
  "include": [
    "src/**/*.worker.ts"
  ]
}
```

However, in our NX workspace, we have a `tsconfig.base.json` file instead of `tsconfig.json`. So, we fixed it.

In the web worker file, we have the `addEventListener` method call, which receives messages from the UI thread to the worker. Notice that we expect to receive the `randomizationCount` property from the UI thread so we can use it in the `generateRandomColor` method from the `utils.ts` file.

The worker file also has a method named `getRandomColorWorker`. This returns a new instance of the worker every time it is called. Since we call it from our user card components, each card gets a new instance of the worker.

Then, in our `UserCardComponent` class, we get a new instance of the worker for each component and use the `ngOnInit` life cycle hook to add an event listener to the worker. This way, whenever the worker sends a message, the user card component can read it—that is, it will get the generated color and assign it to the `randomColor` property. This, in turn, will set the `backgroundColor` of the user card. Notice that the `ngOnInit` life cycle hook only registers a listener for the messages sent by the worker. But first, we have to tell the worker to generate random colors. For this, we use the `ngOnChanges` life cycle hook. In the hook, we observe the value of the user input. If it changes, we send a message to the worker to generate a random color for the specific user card. If you click on the **Update Irineu's Name** button, you will see consecutive logs from and to the user card. Note that moving the color generation to the worker in this way also results in other components not being re-rendered again when we have any of the browser events (click, key up, etc.) triggering. Finally, we use the `ngOnDestroy` life cycle hook to terminate the worker to avoid memory leaks.

Note that in the `ngOnChanges` hook, we are also falling back to the regular usage of the methods in the `utils.ts` file if the browser does not support workers.

See also

- Angular official documentation on web workers: `https://angular.io/guide/web-worker`
- MDN web worker documentation: `https://developer.mozilla.org/en-US/docs/Web/API/Worker`

Using performance budgets for auditing

In today's world, most of the population has a good internet connection to use everyday applications, be it a mobile app or a web app, and it is fascinating how much data we ship to our end users as a business. The amount of JavaScript shipped to users has an ever-increasing trend now, and if you're working on a web app, you might want to use performance budgets to make sure the bundle size doesn't exceed a certain limit. With Angular apps, setting the budget sizes is a breeze. In this recipe, you're going to learn how to use the performance budgets to ensure small bundle sizes for our Angular apps.

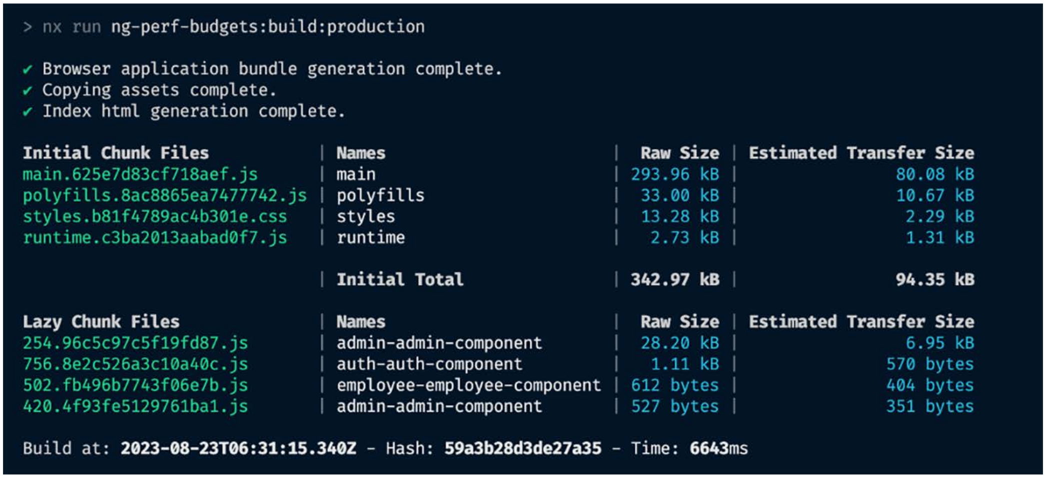
Getting ready

The app that we are going to work with resides in `start/apps/chapter12/ng-perf-budgets` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to build the project:

   ```
   npm run build ng-perf-budgets
   ```

   This should build the app, and you should see the following in the terminal:



```
> nx run ng-perf-budgets:build:production

✔ Browser application bundle generation complete.
✔ Copying assets complete.
✔ Index html generation complete.

Initial Chunk Files           | Names                       |  Raw Size | Estimated Transfer Size
main.625e7d83cf718aef.js      | main                        | 293.96 kB |                80.08 kB
polyfills.8ac8865ea7477742.js | polyfills                   |  33.00 kB |                10.67 kB
styles.b81f4789ac4b301e.css   | styles                      |  13.28 kB |                 2.29 kB
runtime.c3ba2013aabad0f7.js   | runtime                     |   2.73 kB |                 1.31 kB

                              | Initial Total               | 342.97 kB |                94.35 kB

Lazy Chunk Files              | Names                       |  Raw Size | Estimated Transfer Size
254.96c5c97c5f19fd87.js       | admin-admin-component       |  28.20 kB |                 6.95 kB
756.8e2c526a3c10a40c.js       | auth-auth-component         |   1.11 kB |               570 bytes
502.fb496b7743f06e7b.js       | employee-employee-component | 612 bytes |               404 bytes
420.4f93fe5129761ba1.js       | admin-admin-component       | 527 bytes |               351 bytes

Build at: 2023-08-23T06:31:15.340Z - Hash: 59a3b28d3de27a35 - Time: 6643ms
```

Figure 12.30: Build output for production mode, without performance budgets

Notice that the bundle size for the `main.*.js` file is about 294 **Kilobytes** (**KB**) now. And now that we have built the app, let's see the steps of the recipe in the next section.

How to do it...

We have an app that is small in terms of bundle size at the moment. However, this could grow into a huge app with upcoming business requirements. For the sake of this recipe, we'll increase the bundle size deliberately and will then use performance budgets to stop the Angular build from being generated for production if the bundle size exceeds the budget. Let's begin the recipe:

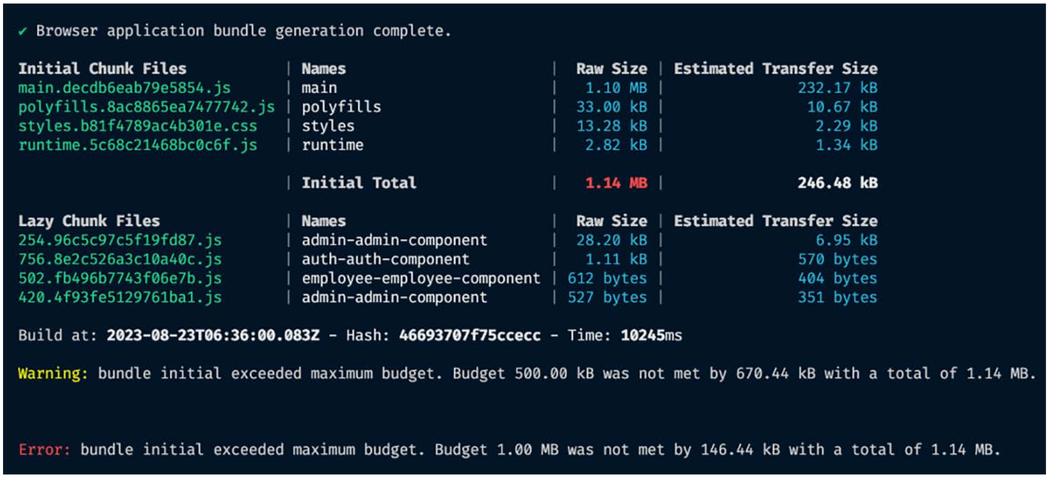
1. Open the `app.component.ts` file and update it, as follows:

```
...
import * as moment from '../lib/moment';
import * as THREE from 'three';
@Component({...})
export class AppComponent {
  ...
  constructor() {
   const scene = new THREE.Scene();
   console.log(moment().format('MMM Do YYYY'));
   console.log(scene);
 }
  ...
}
```

2. Now, build the app again by running the following command from the workspace's root folder:

```
npm run build ng-perf-budgets
```

You should see that the bundle size for the `main.*.js` file is now somewhat around 1.10 **megabytes** (**MB**). This is a huge increase in size compared to the original ~294 KB, as you can see in the following screenshot:



```
✔ Browser application bundle generation complete.

Initial Chunk Files           | Names                        |  Raw Size | Estimated Transfer Size
main.decdb6eab79e5854.js      | main                         |   1.10 MB |               232.17 kB
polyfills.8ac8865ea7477742.js | polyfills                    |  33.00 kB |                10.67 kB
styles.b81f4789ac4b301e.css   | styles                       |  13.28 kB |                 2.29 kB
runtime.5c68c21468bc0c6f.js   | runtime                      |   2.82 kB |                 1.34 kB

                              | Initial Total                |   1.14 MB |               246.48 kB

Lazy Chunk Files              | Names                        |  Raw Size | Estimated Transfer Size
254.96c5c97c5f19fd87.js       | admin-admin-component        |  28.20 kB |                 6.95 kB
756.8e2c526a3c10a40c.js       | auth-auth-component          |   1.11 kB |               570 bytes
502.fb496b7743f06e7b.js       | employee-employee-component  | 612 bytes |               404 bytes
420.4f93fe5129761ba1.js       | admin-admin-component        | 527 bytes |               351 bytes

Build at: 2023-08-23T06:36:00.083Z - Hash: 46693707f75ccecc - Time: 10245ms

Warning: bundle initial exceeded maximum budget. Budget 500.00 kB was not met by 670.44 kB with a total of 1.14 MB.



Error: bundle initial exceeded maximum budget. Budget 1.00 MB was not met by 146.44 kB with a total of 1.14 MB.
```

Figure 12.31: The bundle size for main..js increased to 1.10 MB*

Since we're using NX for our recipes, we already have the budgets set by NX. However, if you had a regular Angular application (without NX), you would update the `angular.json` file to add the budgets.

3. Open the `project.json` file within the `start/apps/chapter12/ng-perf-budgets` folder and update it. The property that we're targeting is `targets.build.configurations.production.budgets`. The updated code should look like this:

```
...
{
  "budgets": [
    {
      "type": "initial",
      "maximumWarning": "800kb",
      "maximumError": "1mb"
    },
     {
     "type": "anyComponentStyle",
     "maximumWarning": "2kb",
     "maximumError": "4kb"
    }
  ]
}
...
```

Note that we have only changed `maximumWarning` from `500kb` to `800kb` for the `initial` bundle.

4. Let's improve our application by not importing the entire libraries in the `app.component.ts` file, and by using the `date-fns` package instead of `moment.js`. Run the following command from the workspace's root folder to install the `date-fns` package:

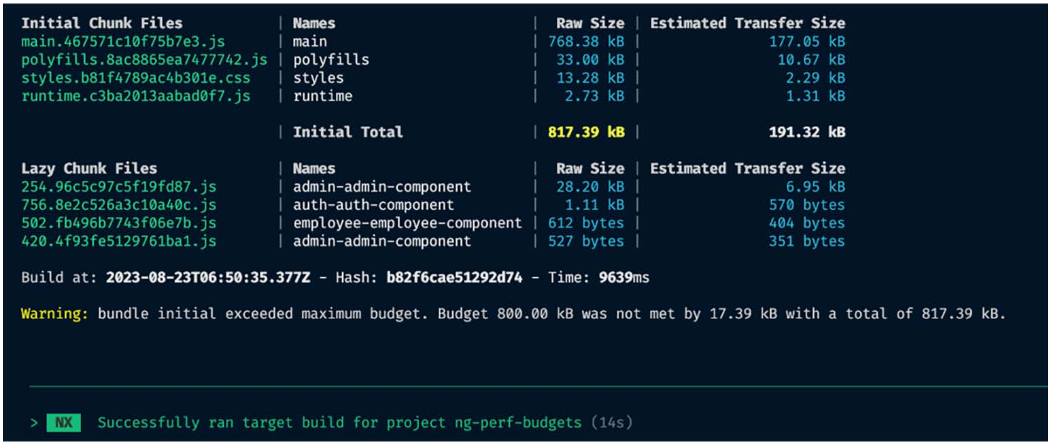
```
cd start && npm install --save date-fns
```

5. Now, update the `app.component.ts` file, as follows:

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { AuthService } from './services/auth.service';
import { format } from 'date-fns';
import { Scene } from 'three';
@Component({...})
export class AppComponent {
  ...
  constructor() {
    const scene = new Scene();
```

```
      console.log(format(new Date(), 'LLL do yyyy'));
      console.log(scene);
    }
    ...
  }
```

6. Run the `npm run build ng-perf-budgets` command again. You should see a decreased bundle size and the build being generated successfully, as follows:



```
Initial Chunk Files            | Names                         |  Raw Size | Estimated Transfer Size
main.467571c10f75b7e3.js       | main                          | 768.38 kB |               177.05 kB
polyfills.8ac8865ea7477742.js  | polyfills                     |  33.00 kB |                10.67 kB
styles.b81f4789ac4b301e.css    | styles                        |  13.28 kB |                 2.29 kB
runtime.c3ba2013aabad0f7.js    | runtime                       |   2.73 kB |                 1.31 kB

                               | Initial Total                 | 817.39 kB |               191.32 kB

Lazy Chunk Files               | Names                         |  Raw Size | Estimated Transfer Size
254.96c5c97c5f19fd87.js        | admin-admin-component         |  28.20 kB |                 6.95 kB
756.8e2c526a3c10a40c.js        | auth-auth-component           |   1.11 kB |               570 bytes
502.fb496b7743f06e7b.js        | employee-employee-component   | 612 bytes |               404 bytes
420.4f93fe5129761ba1.js        | admin-admin-component         | 527 bytes |               351 bytes

Build at: 2023-08-23T06:50:35.377Z - Hash: b82f6cae51292d74 - Time: 9639ms

Warning: bundle initial exceeded maximum budget. Budget 800.00 kB was not met by 17.39 kB with a total of 817.39 kB.


 > NX  Successfully ran target build for project ng-perf-budgets (14s)
```

Figure 12.32: Reduced bundle size after using date-fns and optimized imports

Boom! You just learned how to use performance budgets in Angular. These budgets can be used to throw warnings and errors based on your configuration. Note that the budgets can be modified based on changing business requirements. However, as engineers, we have to be cautious about what we set as performance budgets to not ship JavaScript over a certain limit to the end users.

Since you've finished the recipe, see the next section on how this works.

How it works...

Angular performance budgets are guidelines set for the acceptable limits of various performance metrics in Angular applications. These budgets help ensure that the application's performance remains within acceptable bounds and does not degrade significantly as the code base grows or changes. The most critical performance metric you can work with is the initial bundle size. This is what the user's device will load first as the main set of JavaScript files, and they are eagerly loaded. In this recipe, we have two issues. First, we use `moment.js`, which is a library that is not *tree-shakable*. This means if we import the library, the whole library is included in the final bundle, whereas tree-shakable libraries have *the code not being used* in the application removed by the build tools when you build them. The second problem we introduced is that we included the entire library, `three`, in our component, which is tree-shakable, but our import is inaccurate. We introduced these problems to see the bundle size increase. But as we saw, in NX, we can use any Angular app's `project.json` file to manage the performance budgets. If you were working on an Angular CLI-based application, you would do the same in the `angular.json` file. In practice, you will be using a warning threshold and an error threshold, which ensures you're not shipping huge JavaScript bundles to the end users.

See also

- Performance budgets with the Angular CLI official documentation: `https://web.dev/performance-budgets-with-the-angular-cli/`
- *Boosting Angular App Performance with Performance Budgets*: `https://www.codewithahsan.dev/blog/angular-performance-budgets`

Analyzing bundles with webpack-bundle-analyzer

In the previous recipe, we looked at configuring budgets for our Angular app, and this is useful because you get to know when the overall bundle size exceeds a certain threshold, although you don't get to know how much each part of the code contributes to the final bundles. This is what we call *analyzing* the bundles, and in this recipe, you will learn how to use `webpack-bundle-analyzer` to audit the bundle sizes and the factors contributing to them.

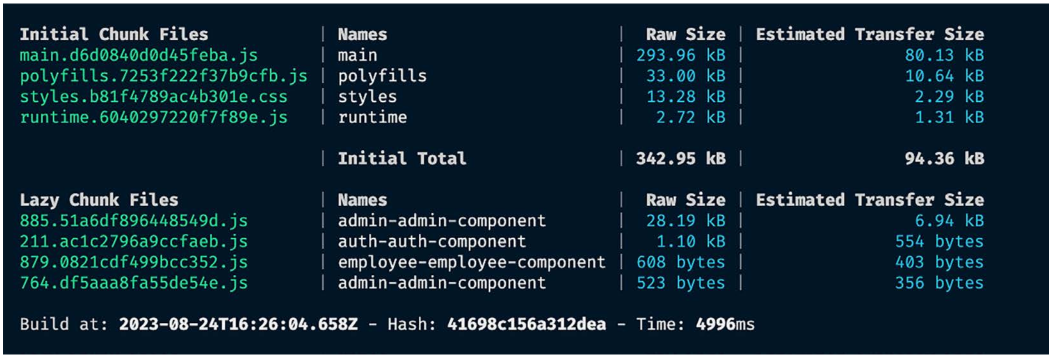
Getting ready

The app that we are going to work with resides in `start/apps/chapter12/ng-perf-wba` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the code repository directory, and run the following command to build the project:

```
npm run build ng-perf-wba
```

This should build the app, and you should see the following in the terminal:



```
Initial Chunk Files           | Names                         |  Raw Size | Estimated Transfer Size
main.d6d0840d0d45feba.js      | main                          | 293.96 kB |                80.13 kB
polyfills.7253f222f37b9cfb.js | polyfills                     |  33.00 kB |                10.64 kB
styles.b81f4789ac4b301e.css   | styles                        |  13.28 kB |                 2.29 kB
runtime.6040297220f7f89e.js   | runtime                       |   2.72 kB |                 1.31 kB

                              | Initial Total                 | 342.95 kB |                94.36 kB

Lazy Chunk Files              | Names                         |  Raw Size | Estimated Transfer Size
885.51a6df896448549d.js       | admin-admin-component         |  28.19 kB |                 6.94 kB
211.ac1c2796a9ccfaeb.js       | auth-auth-component           |   1.10 kB |               554 bytes
879.0821cdf499bcc352.js       | employee-employee-component   | 608 bytes |               403 bytes
764.df5aaa8fa55de54e.js       | admin-admin-component         | 523 bytes |               356 bytes

Build at: 2023-08-24T16:26:04.658Z - Hash: 41698c156a312dea - Time: 4996ms
```

Figure 12.33: The ng-perf-wba app running at http://localhost:4200

Now that we have built the app, let's see the steps of the recipe in the next section.

How to do it...

We have an app that is small in terms of bundle size at the moment. However, this could grow into a huge app with upcoming business requirements. For the sake of this recipe, we'll increase the bundle size deliberately and will then use `webpack-bundle-analyzer` to observe the packages contributing to the large bundle sizes. Let's begin the recipe:

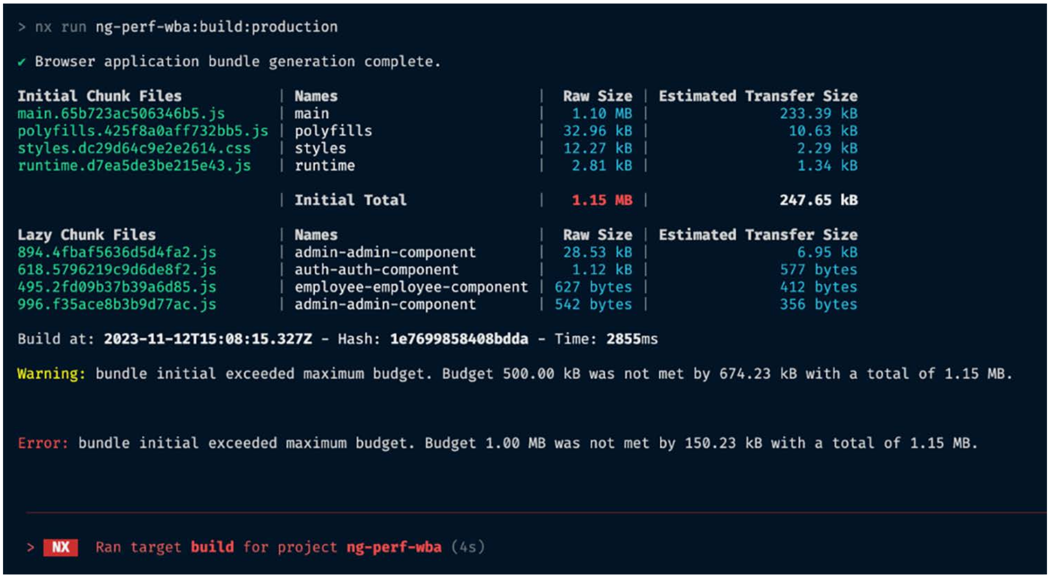
1. Open the `app.component.ts` file and update it, as follows:

```
...
import * as moment from '../lib/moment';
import * as THREE from 'three';
@Component({...})
export class AppComponent {
  ...
  constructor() {
    const scene = new THREE.Scene();
    console.log(moment().format('MMM Do YYYY'));
    console.log(scene);
  }
  ...
}
```

2. Now, build the app again by running the following command from the workspace root:

```
npm run build ng-perf-wba
```

You should see that the bundle size for the `main.*.js` file is now somewhat around 1.10 MB. This is a huge increase in size compared to the original ~294 KB. As a result, the **Initial Total** size became 1.15 MB and the build failed, as you can see in the following screenshot:



```
> nx run ng-perf-wba:build:production

✔ Browser application bundle generation complete.

Initial Chunk Files           | Names                        |  Raw Size | Estimated Transfer Size
main.65b723ac506346b5.js      | main                         |   1.10 MB |               233.39 kB
polyfills.425f8a0aff732bb5.js | polyfills                    |  32.96 kB |                10.63 kB
styles.dc29d64c9e2e2614.css   | styles                       |  12.27 kB |                 2.29 kB
runtime.d7ea5de3be215e43.js   | runtime                      |   2.81 kB |                 1.34 kB

                              | Initial Total                |   1.15 MB |               247.65 kB

Lazy Chunk Files              | Names                        |  Raw Size | Estimated Transfer Size
894.4fbaf5636d5d4fa2.js       | admin-admin-component        |  28.53 kB |                 6.95 kB
618.5796219c9d6de8f2.js       | auth-auth-component          |   1.12 kB |               577 bytes
495.2fd09b37b39a6d85.js       | employee-employee-component  | 627 bytes |               412 bytes
996.f35ace8b3b9d77ac.js       | admin-admin-component        | 542 bytes |               356 bytes

Build at: 2023-11-12T15:08:15.327Z - Hash: 1e7699858408bdda - Time: 2855ms

Warning: bundle initial exceeded maximum budget. Budget 500.00 kB was not met by 674.23 kB with a total of 1.15 MB.


Error: bundle initial exceeded maximum budget. Budget 1.00 MB was not met by 150.23 kB with a total of 1.15 MB.



 >  NX   Ran target build for project ng-perf-wba (4s)
```

Figure 12.34: The build failing due to an increased bundle size

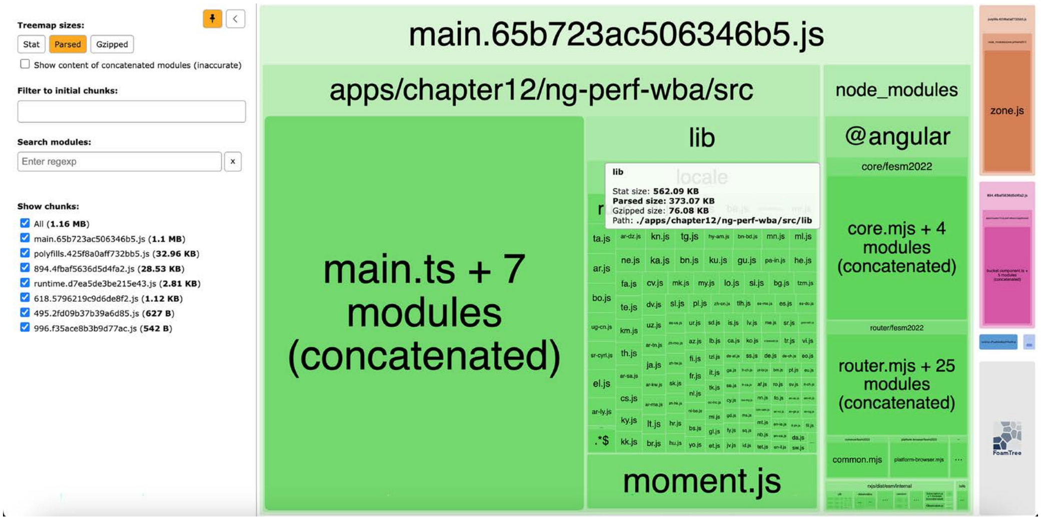
3. We'll first create a build with the `stats.json` file, which contains the bundle information in the JSON format. To do that, run the following command from the workspace root:

```
npm run build ng-perf-wba with-stats
```

4. Now, run the following command from the workspace root to have the `stats.json` file read by the `webpack-bundle-analyzer`, as follows:

```
npx webpack-bundle-analyzer ./start/dist/apps/chapter12/ng-perf-wba/
stats.json
```

This will spin up a server with the bundle analysis. You should see a new tab opened in your default browser, and it should look like this:


Treemap sizes:
Stat
Parsed
Gzipped
Filter to initial chunks:
Search modules:
Enter regexp
×
main.65b723ac506346b5.js
apps/chapter12/ng-perf-wba/src
main.ts + 7 modules (concatenated)
lib
locale
moment.js
node_modules
@angular
core/fesm2022
core.mjs + 4 modules (concatenated)
router/fesm2022
router.mjs + 25 modules (concatenated)
common.mjs
platform-browser.mjs
zone.js
FoamTree

☐ Show content of concatenated modules (inaccurate)

Show chunks:

☑ All (**1.16 MB**)
☑ main.65b723ac506346b5.js (**1.1 MB**)
☑ polyfills.425f8a0aff732bb5.js (**32.96 KB**)
☑ 894.4fbaf5636d5d4fa2.js (**28.53 KB**)
☑ runtime.d7ea5de3be215e43.js (**2.81 KB**)
☑ 618.5796219c9d6de8f2.js (**1.12 KB**)
☑ 495.2fd09b37b39a6d85.js (**627 B**)
☑ 996.f35ace8b3b9d77ac.js (**542 B**)

lib

Stat size: **562.09 KB**
Parsed size: 373.07 KB
Gzipped size: **76.08 KB**
Path: **./apps/chapter12/ng-perf-wba/src/lib**

Figure 12.35: Bundle analysis using webpack-bundle-analyzer

5. Notice that the `lib` folder takes a huge portion of the bundle size— around 562 KB, which you can check by just doing a mouseover on the **lib** box. The overall bundle size is 1.16 MB. Let's try to optimize the bundle size. Let's install the `date-fns` package so that we can use it instead of `moment.js`. Run the following command from your project root:

```
cd start && npm install --save date-fns
```

6. Now, update the `app.component.ts` file to use the `date-fns` package's `format` method instead of using the `moment().format` method. We'll also just import the `Scene` class from the `Three.js` package instead of importing the whole library. The code should look like this:

```
...
import { AuthService } from './services/auth.service';
import { format } from 'date-fns';
import { Scene } from 'three';
@Component({...})
```

```
export class AppComponent {
  ...
  constructor() {
    const scene = new Scene();
    console.log(format(new Date(), 'LLL do yyyy'));
    console.log(scene);
  }
  ...
}
```

7. Repeat *step 3* and *step 4* to build the app again and to analyze via `webpack-bundle-analyzer`.

   Once `webpack-bundle-analyzer` runs, you should see the analysis, as shown in the following screenshot. Notice that we don't have the `moment.js` file or the `lib` block anymore, and the overall bundle size has reduced from 1.16 MB to about 835 KB:

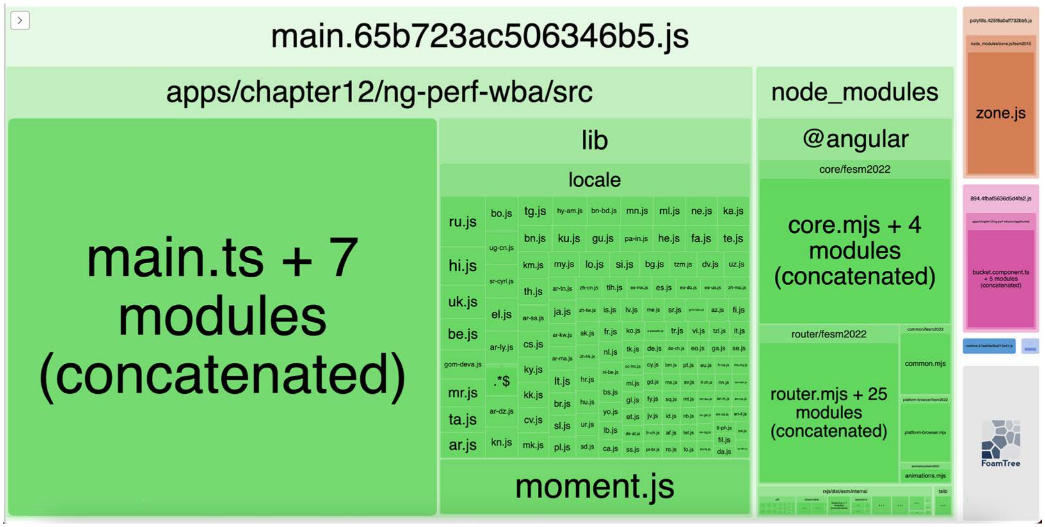

main.65b723ac506346b5.js
apps/chapter12/ng-perf-wba/src
main.ts + 7 modules (concatenated)
lib
locale
moment.js
node_modules
@angular
core/fesm2022
core.mjs + 4 modules (concatenated)
router/fesm2022
router.mjs + 25 modules (concatenated)
common.mjs
animations.mjs
zone.js
FoamTree

Figure 12.36: Bundle analysis after optimization

Woohoo! You now know how to use the `webpack-bundle-analyzer` package to audit bundle sizes in Angular applications. This is a great way of improving overall performance because you can identify the chunks causing the increase in the bundle size and then optimize the bundles. If you run `npm run serve ng-perf-wba` from the workspace root before and after the optimization, you will see the same console logs, which show that we kept the existing functionality and optimized the bundles anyway.

See also

- Getting started with webpack: `https://webpack.js.org/guides/getting-started/`
- The `webpack-bundle-analyzer` GitHub repository: `https://github.com/webpack-contrib/webpack-bundle-analyzer`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

13

Building PWAs with Angular

Progressive Web Apps (**PWAs**) are far more than just web applications; they're the next evolution in web technology. Combining the best of web and mobile applications, PWAs provide an unparalleled user experience, even in less-than-ideal network conditions. But what makes them truly compelling is their graceful degradation — while they harness the full power of modern browsers, they also ensure a seamless, core experience in older browsers.

In this chapter, we dive deep into the world of PWAs through the lens of Angular. Angular has built-in PWA support that makes it an ideal candidate for crafting resilient and performant web applications. You'll learn how to make your apps installable, capable, fast, and reliable building them as progressive web apps. The following are the recipes we're going to cover in this chapter:

- Converting an existing Angular app into a PWA with the Angular CLI
- Modifying the theme color for your PWA
- Using dark mode in your PWA
- Providing a custom installable experience in your PWA
- Precaching requests using an Angular service worker
- Creating an app shell for your PWA

Technical requirements

For the recipes in this chapter, ensure your setup is complete as per the 'Technical Requirements' in the 'Angular-Cookbook-2E' GitHub repository. For setup details, visit: `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/docs/technical-requirements.md`. The starter code for this chapter is located at `https://github.com/PacktPublishing/Angular-Cookbook-2E/tree/main/start/apps/chapter13`.

Converting an existing Angular app into a PWA with the Angular CLI

A **PWA** comprises several intriguing components, with two standout features being the service worker and the web manifest file. The service worker plays a vital role in caching static resources and handling caching requests.

Meanwhile, the web manifest file holds essential information such as app icons and the app's theme color. In this guide, we'll transform an existing Angular application into a PWA. These principles are also applicable if you were to start a new Angular app from scratch. Throughout this walkthrough, we'll convert an existing Angular app, highlighting the changes made and showcasing the transformation process using the `@angular/pwa` package. This package not only enables PWA functionality but also facilitates effective static resource caching.

Getting ready

The app that we are going to work with resides in `start/apps/chapter13/ng-pwa-conversion` inside the cloned repository. This is not part of our NX workspace and is rather a **standalone Angular** app with its own `package.json`, `node_modules`, etc. Perform the following steps to get started:

1. Open the code repository in your code editor.
2. Open the terminal and navigate to the cloned code repository folder.
3. Navigate to `start/apps/chapter13/ng-pwa-conversion` and run the following commands to serve the project:

   ```
   npm run build && npx http-server dist/ng-pwa-conversion
   ```

   Open a new tab in the browser and navigate to `http://localhost:8080`. You should see the following:

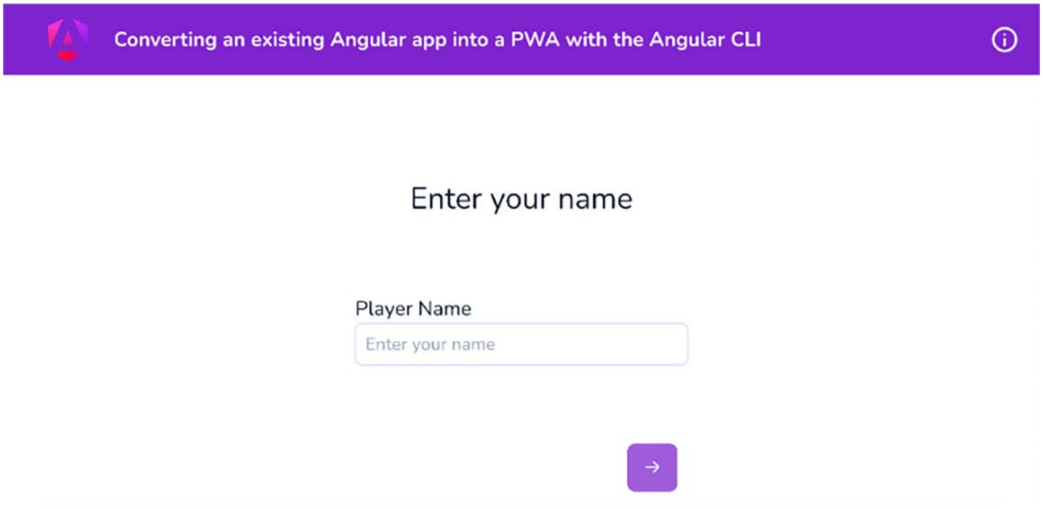

Converting an existing Angular app into a PWA with the Angular CLI
Enter your name
Player Name
Enter your name

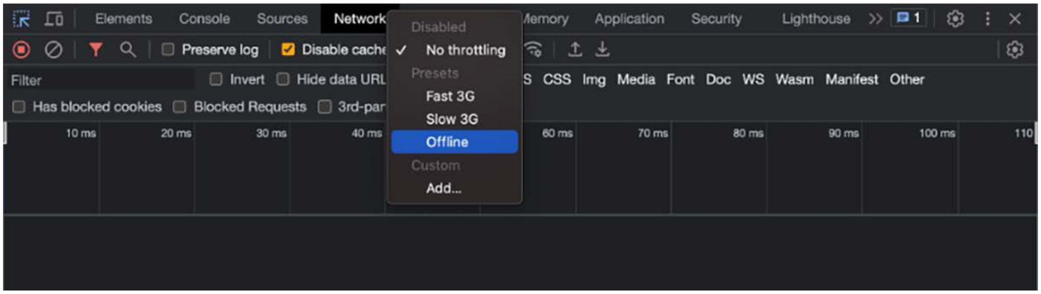
Figure 13.1: ng-pwa-conversion app running on http://localhost:8080

Now that we have the app running locally, let's see the steps of the recipe in the next section.

How to do it...

The app we're working with is a game built using the Angular CDK in *Chapter 9*, *Angular and the Angular CDK*. You can enter your name, then guess the dice's next value, and get a score leaderboard. The value of the leaderboard is persisted using the local storage. But the app is not a PWA yet. Let's convert it into a PWA:

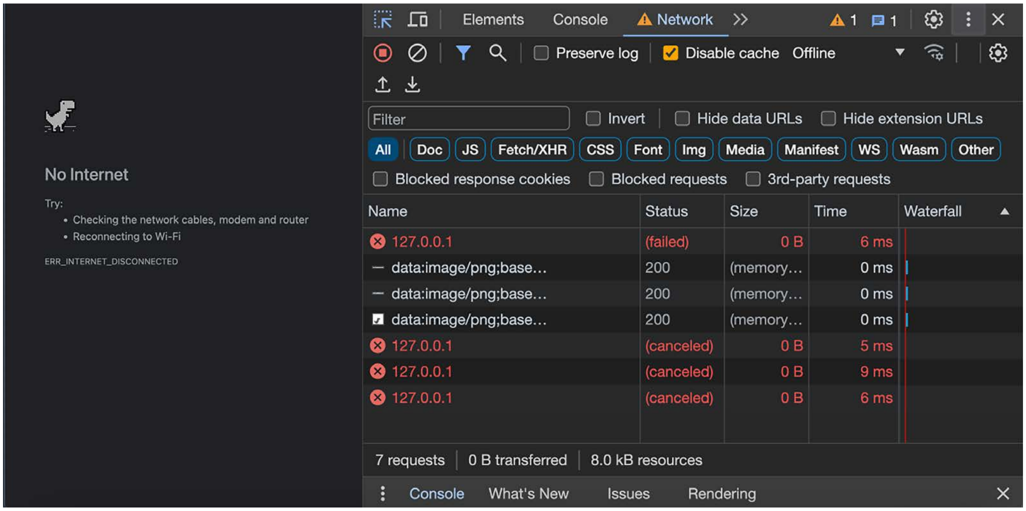
1. First, let's see if our application works offline at all, because that is one of the traits of PWAs. Open **Chrome Dev Tools** for the app. Go to the **Network** tab and change **Throttling** to **Offline**, as follows:



Elements
Console
Sources
Network
Memory
Application
Security
Lighthouse
1
☐ Preserve log
☑ Disable cache
Filter
☐ Invert
☐ Hide data URL
☐ Has blocked cookies
☐ Blocked Requests
☐ 3rd-par
S CSS Img Media Font Doc WS Wasm Manifest Other
Disabled
✓ No throttling
Presets
Fast 3G
Slow 3G
Offline
Custom
Add...
10 ms
20 ms
30 ms
40 ms
60 ms
70 ms
80 ms
90 ms
100 ms
110

Figure 13.2: Changing network throttling to Offline to see the offline experience

2. Make sure that the **Disable cache** option is ticked too.
3. Now stop the `http` server by exiting the process from your terminal. Once done, refresh the app's page. You should see that the app doesn't work anymore, as shown in the following figure:



Elements
Console
Network
1
1
☐ Preserve log
☑ Disable cache
Offline
Filter
☐ Invert
☐ Hide data URLs
☐ Hide extension URLs
All
Doc
JS
Fetch/XHR
CSS
Font
Img
Media
Manifest
WS
Wasm
Other
☐ Blocked response cookies
☐ Blocked requests
☐ 3rd-party requests
Console
What's New
Issues
Rendering

No Internet

Try:

- Checking the network cables, modem and router
- Reconnecting to Wi-Fi

ERR_INTERNET_DISCONNECTED

Name	Status	Size	Time	Waterfall
127.0.0.1	(failed)	0 B	6 ms	
data:image/png;base...	200	(memory...	0 ms	
data:image/png;base...	200	(memory...	0 ms	
data:image/png;base...	200	(memory...	0 ms	
127.0.0.1	(canceled)	0 B	5 ms	
127.0.0.1	(canceled)	0 B	9 ms	
127.0.0.1	(canceled)	0 B	6 ms	

7 requests | 0 B transferred | 8.0 kB resources

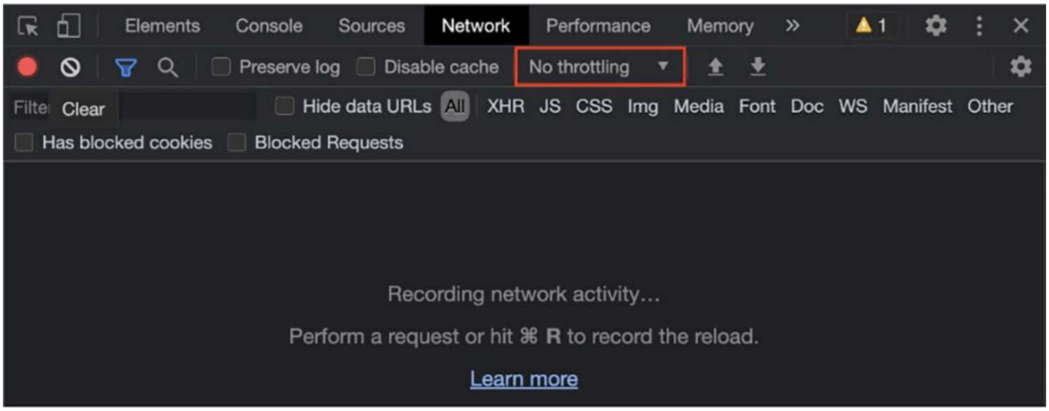
Figure 13.3: App not working offline

4. To convert this app into a PWA, open a new terminal window/tab and make sure you're inside the `start/apps/chapter13/ng-pwa-conversion` folder. Once inside, run the following command:

```
ng add @angular/pwa
```

You should see a bunch of files created and updated as the process from the command finishes.

5. Now build and serve the app again by running `npm run build && npx http-server dist/ng-pwa-conversion`. Once done, navigate to `http://localhost:8080`.
6. Now make sure you have turned off throttling by switching to the **Network** tab and setting **No throttling** as the selection option, as shown in *Figure 13.4*. Also, notice that the **Disable cache** option is turned off:


Elements
Console
Sources
Network
Performance
Memory
1
Preserve log
Disable cache
No throttling
Filte
Clear
Hide data URLs
All
XHR
JS
CSS
Img
Media
Font
Doc
WS
Manifest
Other
Has blocked cookies
Blocked Requests

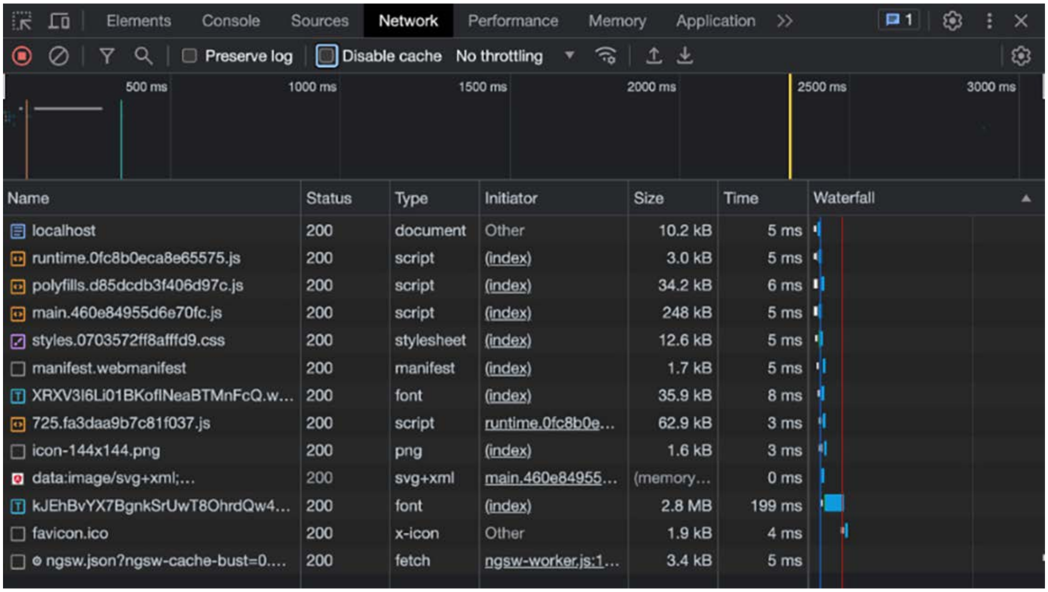
Recording network activity…

Perform a request or hit ⌘ R to record the reload.

Learn more

Figure 13.4: Turning off network throttling

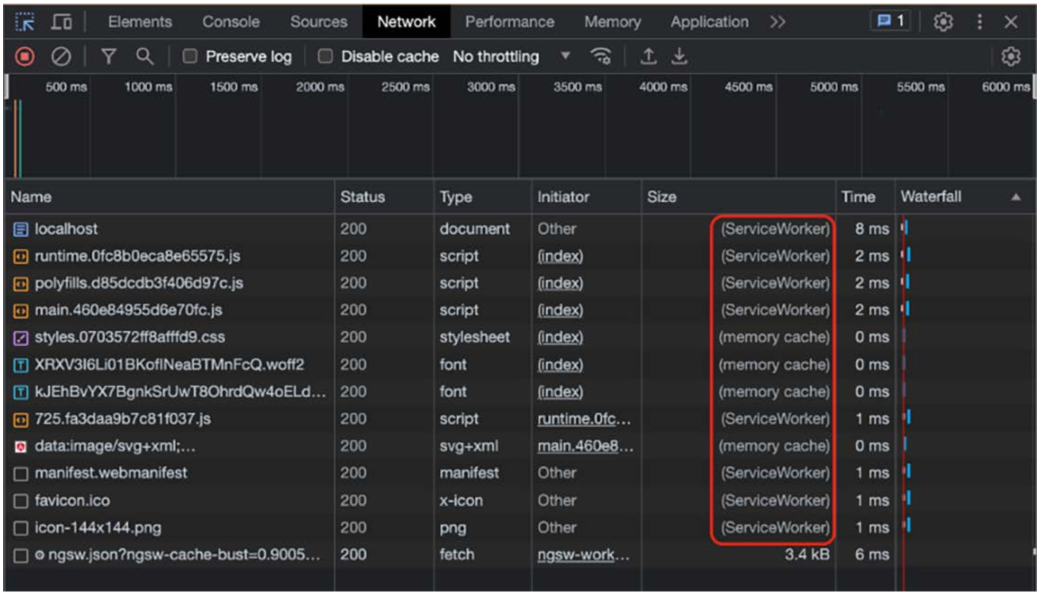
7. Now refresh the app once. You should see the app working and the network logs showing that assets such as JavaScript files were loaded from the server, as shown in *Figure 13.5*:


Elements
Console
Sources
Network
Performance
Memory
Application
1
Preserve log
Disable cache
No throttling
500 ms
1000 ms
1500 ms
2000 ms
2500 ms
3000 ms

Name	Status	Type	Initiator	Size	Time	Waterfall
localhost	200	document	Other	10.2 kB	5 ms	
runtime.0fc8b0eca8e65575.js	200	script	(index)	3.0 kB	5 ms	
polyfills.d85dcdb3f406d97c.js	200	script	(index)	34.2 kB	6 ms	
main.460e84955d6e70fc.js	200	script	(index)	248 kB	5 ms	
styles.0703572ff8afffd9.css	200	stylesheet	(index)	12.6 kB	5 ms	
manifest.webmanifest	200	manifest	(index)	1.7 kB	5 ms	
XRXV3I6Li01BKofINeaBTMnFcQ.w…	200	font	(index)	35.9 kB	8 ms	
725.fa3daa9b7c81f037.js	200	script	runtime.0fc8b0e…	62.9 kB	3 ms	
icon-144x144.png	200	png	(index)	1.6 kB	3 ms	
data:image/svg+xml;…	200	svg+xml	main.460e84955…	(memory…	0 ms	
kJEhBvYX7BgnkSrUwT8OhrdQw4…	200	font	(index)	2.8 MB	199 ms	
favicon.ico	200	x-icon	Other	1.9 kB	4 ms	
⚙ ngsw.json?ngsw-cache-bust=0.…	200	fetch	ngsw-worker.js:1…	3.4 kB	5 ms	

Figure 13.5: Assets downloaded from the source (Angular server)

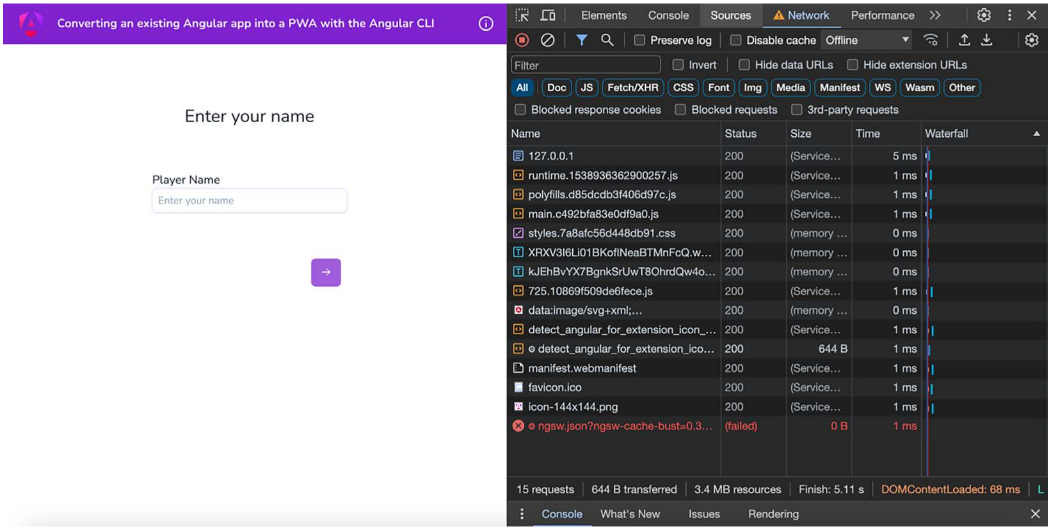
8. Now, refresh the app once again and you'll see that the same assets are now downloaded from the cache using the service worker, as shown in *Figure 13.6*:


Elements
Console
Sources
Network
Performance
Memory
Application
Preserve log
Disable cache
No throttling
500 ms
1000 ms
1500 ms
2000 ms
2500 ms
3000 ms
3500 ms
4000 ms
4500 ms
5000 ms
5500 ms
6000 ms

Name	Status	Type	Initiator	Size	Time
localhost	200	document	Other	(ServiceWorker)	8 ms
runtime.0fc8b0eca8e65575.js	200	script	(index)	(ServiceWorker)	2 ms
polyfills.d85dcdb3f406d97c.js	200	script	(index)	(ServiceWorker)	2 ms
main.460e84955d6e70fc.js	200	script	(index)	(ServiceWorker)	2 ms
styles.0703572ff8afffd9.css	200	stylesheet	(index)	(memory cache)	0 ms
XRXV3I6Li01BKofINeaBTMnFcQ.woff2	200	font	(index)	(memory cache)	0 ms
kJEhBvYX7BgnkSrUwT8OhrdQw4oELd…	200	font	(index)	(memory cache)	0 ms
725.fa3daa9b7c81f037.js	200	script	runtime.0fc…	(ServiceWorker)	1 ms
data:image/svg+xml;…	200	svg+xml	main.460e8…	(memory cache)	0 ms
manifest.webmanifest	200	manifest	Other	(ServiceWorker)	1 ms
favicon.ico	200	x-icon	Other	(ServiceWorker)	1 ms
icon-144x144.png	200	png	Other	(ServiceWorker)	1 ms
⚙ ngsw.json?ngsw-cache-bust=0.9005…	200	fetch	ngsw-work…	3.4 kB	6 ms

Figure 13.6: Assets downloaded from the cache using the service worker

9. Now is the moment we've been waiting for. Change the network throttling back to **Offline** to go into the **Offline** mode and refresh the app. You should still see the app working in **Offline** mode because of the service worker, as shown in *Figure 13.7*:

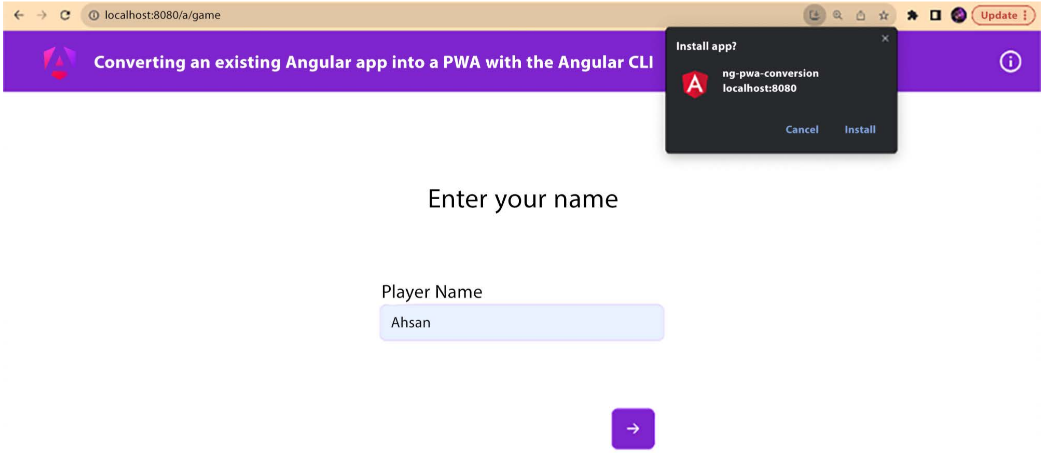

Converting an existing Angular app into a PWA with the Angular CLI
Enter your name
Player Name
Enter your name
Elements
Console
Sources
Network
Performance
Preserve log
Disable cache
Offline
Filter
Invert
Hide data URLs
Hide extension URLs
All
Doc
JS
Fetch/XHR
CSS
Font
Img
Media
Manifest
WS
Wasm
Other
Blocked response cookies
Blocked requests
3rd-party requests
Console
What's New
Issues
Rendering

Name	Status	Size	Time
127.0.0.1	200	(Service…	5 ms
runtime.1538936362900257.js	200	(Service…	1 ms
polyfills.d85dcdb3f406d97c.js	200	(Service…	1 ms
main.c492bfa83e0df9a0.js	200	(Service…	1 ms
styles.7a8afc56d448db91.css	200	(memory …	0 ms
XRXV3I6Li01BKofINeaBTMnFcQ.w…	200	(memory …	0 ms
kJEhBvYX7BgnkSrUwT8OhrdQw4o…	200	(memory …	0 ms
725.10869f509de6fece.js	200	(Service…	1 ms
data:image/svg+xml;…	200	(memory …	0 ms
detect_angular_for_extension_icon_…	200	(Service…	1 ms
⚙ detect_angular_for_extension_ico…	200	644 B	1 ms
manifest.webmanifest	200	(Service…	1 ms
favicon.ico	200	(Service…	1 ms
icon-144x144.png	200	(Service…	1 ms
⚙ ngsw.json?ngsw-cache-bust=0.3…	(failed)	0 B	1 ms

15 requests | 644 B transferred | 3.4 MB resources | Finish: 5.11 s | DOMContentLoaded: 68 ms

Figure 13.7: Angular app working offline as a PWA using a service worker

10. Now you can install this PWA on your machine. Since I'm using a MacBook, it is installed as a Mac app. If you're using Chrome, the installation option should be around the address bar, as shown in *Figure 13.8*:



localhost:8080/a/game

Converting an existing Angular app into a PWA with the Angular CLI

Install app?

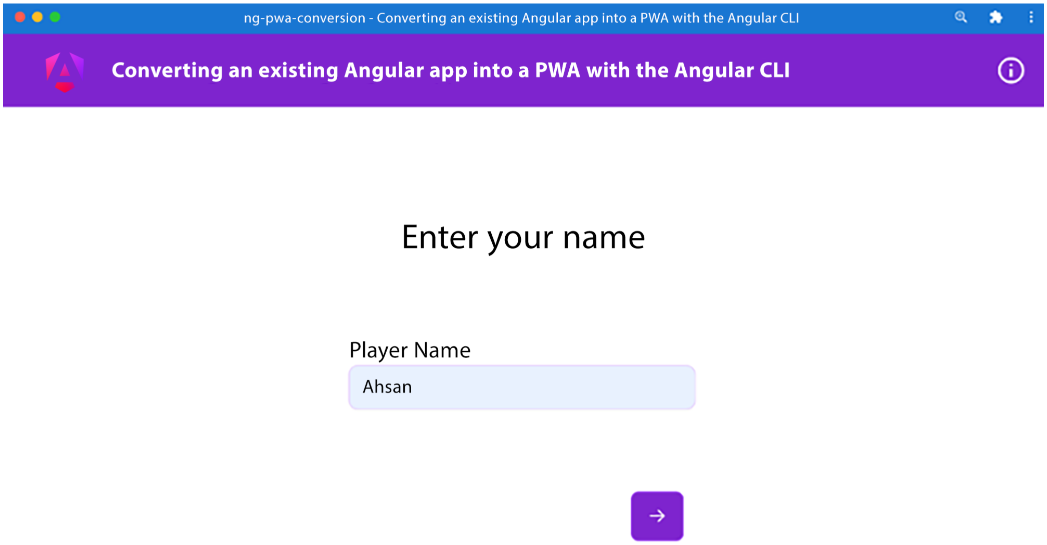
ng-pwa-conversion
localhost:8080

Enter your name

Player Name: Ahsan

Figure 13.8: Installing the Angular PWA from Chrome

Kaboom! Just by using the `@angular/pwa` package, with zero configuration done ourselves, we converted our existing Angular app into a PWA. We are now able to run our application offline, and we can install it as a PWA on our devices. See *Figure 13.9* to see how the app looks—just like a native app on Mac OS X:



ng-pwa-conversion - Converting an existing Angular app into a PWA with the Angular CLI

Converting an existing Angular app into a PWA with the Angular CLI

Enter your name

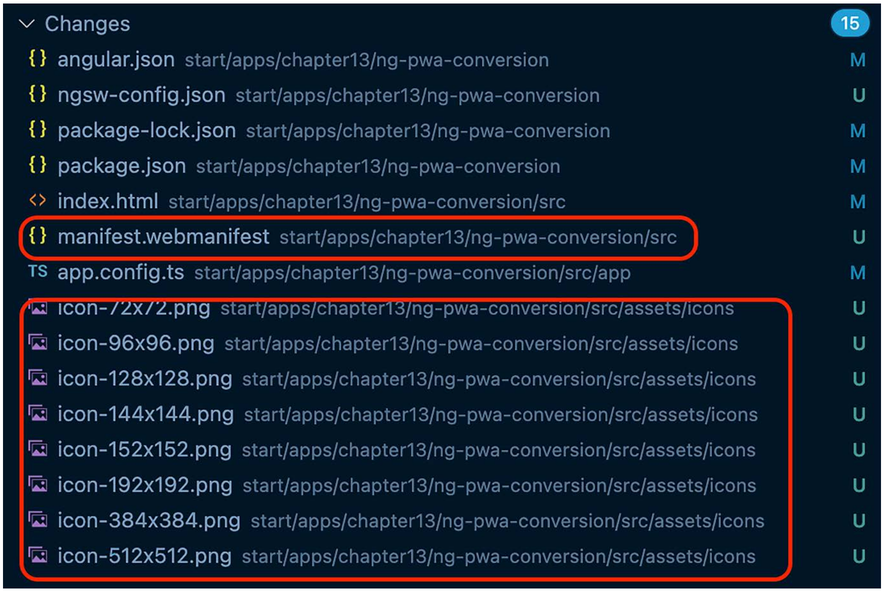
Player Name: Ahsan

Figure 13.9: How our Angular PWA looks as a native app on Mac OS X

Cool, *right?* Now that you know how to build a PWA with the Angular CLI, see the next section to understand how it works.

How it works...

The Angular core team and the community have done an amazing job with the `@angular/pwa` package and, in general, with the `ng add` command, which allows us to add different packages to our applications using Angular schematics. In this recipe, when we run `ng add @angular/pwa`, it uses schematics to generate the app icons along with the web app manifest. If you look at the changed files, you can see the new files, as shown in *Figure 13.10*:



Changes 15

angular.json start/apps/chapter13/ng-pwa-conversion M
ngsw-config.json start/apps/chapter13/ng-pwa-conversion U
package-lock.json start/apps/chapter13/ng-pwa-conversion M
package.json start/apps/chapter13/ng-pwa-conversion M
index.html start/apps/chapter13/ng-pwa-conversion/src M
manifest.webmanifest start/apps/chapter13/ng-pwa-conversion/src U
app.config.ts start/apps/chapter13/ng-pwa-conversion/src/app M
icon-72x72.png start/apps/chapter13/ng-pwa-conversion/src/assets/icons U
icon-96x96.png start/apps/chapter13/ng-pwa-conversion/src/assets/icons U
icon-128x128.png start/apps/chapter13/ng-pwa-conversion/src/assets/icons U
icon-144x144.png start/apps/chapter13/ng-pwa-conversion/src/assets/icons U
icon-152x152.png start/apps/chapter13/ng-pwa-conversion/src/assets/icons U
icon-192x192.png start/apps/chapter13/ng-pwa-conversion/src/assets/icons U
icon-384x384.png start/apps/chapter13/ng-pwa-conversion/src/assets/icons U
icon-512x512.png start/apps/chapter13/ng-pwa-conversion/src/assets/icons U

Figure 13.10: Web manifest file and the app icon files

The `manifest.webmanifest` file is a file that contains a JSON object. This object defines the manifest for the PWA and contains some information. The information includes the name of the app, the short name, the theme color, and the configuration for different icons, for different devices. Imagine this PWA is installed on your Android phone. You need an icon in your home drawer to tap on to open the app. This file holds the information regarding which icon to use based on different device sizes.

We also see the file `ngsw-config.json`, which contains the configuration for the service worker. Behind the scenes, while the `ng add` command is running the schematics, it also installs the `@angular/service-worker` package in our project. If you open the `app.config.ts` file, you'll see the code to register our service worker as follows:

```
...
import { provideServiceWorker } from '@angular/service-worker';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(appRoutes,
        withEnabledBlockingInitialNavigation(), withHashLocation()),
        provideServiceWorker('ngsw-worker.js', {
            enabled: !isDevMode(),
            registrationStrategy: 'registerWhenStable:30000'
        })],
};
```

Let's break down the code to understand what's happening here. We're creating the configuration for our standalone application here along with some providers. We have used the `provideRouter` method to provide all the routes for the application already. In the recipe, we add the `provideServiceWorker` method to register a new service worker for us. It takes two arguments:

- The first argument is the filename of the service worker script, `'ngsw-worker.js'`. This is the built-in Angular service worker that handles the caching and other offline behaviors.
- The second argument is a configuration object for the service worker. Let's review the properties of this configuration object:
    - `enabled: !isDevMode()` tells the application to enable the service worker only when the application is not in development mode.
    - `registrationStrategy:'registerWhenStable:3000'` specifies the strategy for registering the service worker. In this case, the service worker will be registered when the application is stable (no ongoing tasks) after 30 seconds.

The code registers a new service worker file named `ngsw-worker.js`. This file uses the configuration from the `ngsw-config.json` file to decide which resource to cache and using which strategies. This is linked in the `angular.json` file using the `ngswConfigPath` property. Note that the application we have is a standalone application. This means we don't have an `NgModule` for the application's bootstrapping process. If this was rather a non-standalone application, we would see the changes in the `app.module.ts` file.

Now that you know how the recipe works, see the next section for further reading.

See also

- Angular service worker intro: `https://angular.io/guide/service-worker-intro`
- What are PWAs?: `https://web.dev/what-are-pwas/`

Modifying the theme color for your PWA

In the previous recipe, we learned how to convert an Angular app into a PWA. When we do so, the `@angular/pwa` package creates the web app manifest file with the default theme color, as shown in *Figure 13.9*. However, almost every web app has its own branding and style. If you want to theme your PWA's title bar according to your branding, this is the recipe for you. We'll learn how to modify the web app manifest file to customize the PWA's theme color.

Getting ready

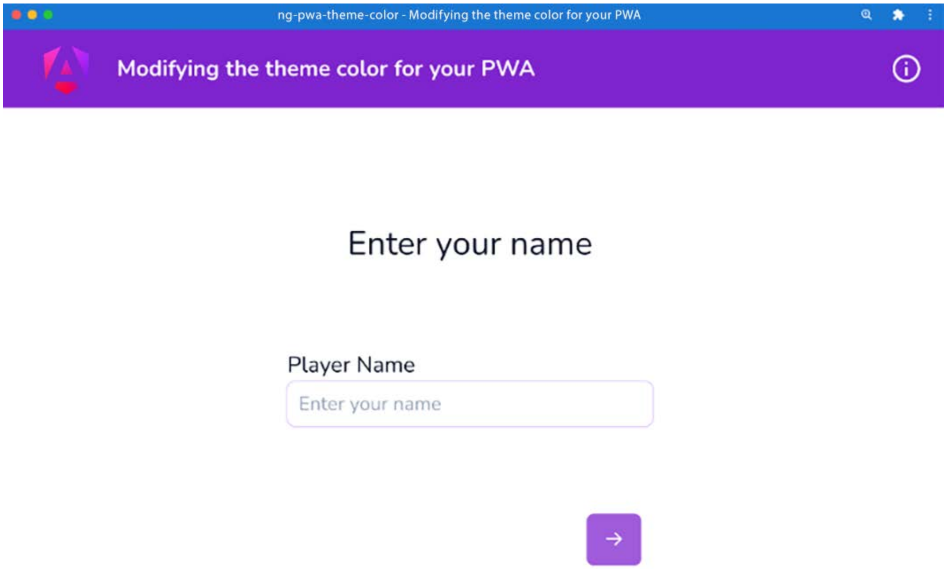
The app that we are going to work with resides in `start/apps/chapter13/ng-pwa-theme-color` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the cloned code repository folder, and run the following command (from the workspace's root folder) to serve the project in production mode:

   ```
   npm run serve:static chapter13 ng-pwa-theme-color 5300
   ```

   This should open the app in a new browser tab at `https://localhost:5300`.
3. Install the application as shown in *Figure 13.8*.

   This should open the app in a native OS window, and you should see the following:



ng-pwa-theme-color - Modifying the theme color for your PWA

Modifying the theme color for your PWA

Enter your name

Player Name

Enter your name

Figure 13.11: ng-pwa-theme-color app running as a PWA

Now that we have the app running, let's see the steps of the recipe in the next section.

How to do it…

As you can see in *Figure 13.11*, the header of the app has a bit of a different color than the app's native header (or toolbar). Due to this difference, the app looks a bit weird. We'll modify the web app manifest to update the theme color. Let's get started:

1. Open the `src/manifest.webmanifest` file in your editor and change the theme color as follows:

```
{
  "name": "ng-pwa-theme-color",
  "short_name": "ng-pwa-theme-color",
  "theme_color": "#7E22CE",
  ...
}
```

2. We also have `theme-color` set in our `index.html` file. By default, that has precedence over the web app manifest file. Therefore, we need to update it. Open the `index.html` file and update it as follows:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    ...
    <link rel="manifest" href="manifest.webmanifest">
  <meta name="theme-color" content="#7E22CE">
  </head>
  <body>
    ...
  </body>
</html>
```

3. Open the PWA again and uninstall it, as shown in *Figure 13.12*. Make sure to check the box that says **Also clear data from Chrome (...)** when prompted:

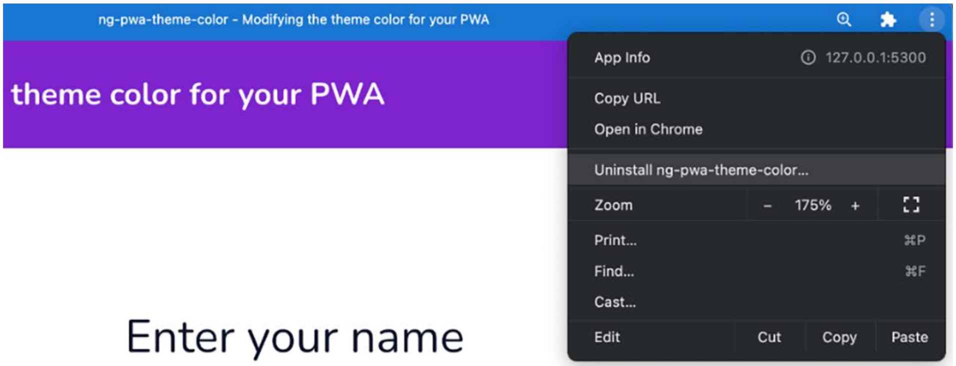

ng-pwa-theme-color - Modifying the theme color for your PWA
theme color for your PWA
Enter your name
App Info
127.0.0.1:5300
Copy URL
Open in Chrome
Uninstall ng-pwa-theme-color...
Zoom
–
175%
+
Print...
⌘P
Find...
⌘F
Cast...
Edit
Cut
Copy
Paste

Figure 13.12: Uninstalling the ng-pwa-theme-color app

4. Now, build the app again using the following commands:

```
npm run build ng-pwa-theme-color && npm run serve:static chapter13 ng-
pwa-theme-color 5300
```

5. Now go to `http://localhost:5300` and install the app again as a PWA, as shown in *Figure 13.8*.
6. The PWA should already be opened. If not, open it from your applications and you should see the updated theme color, as shown in *Figure 13.13*:

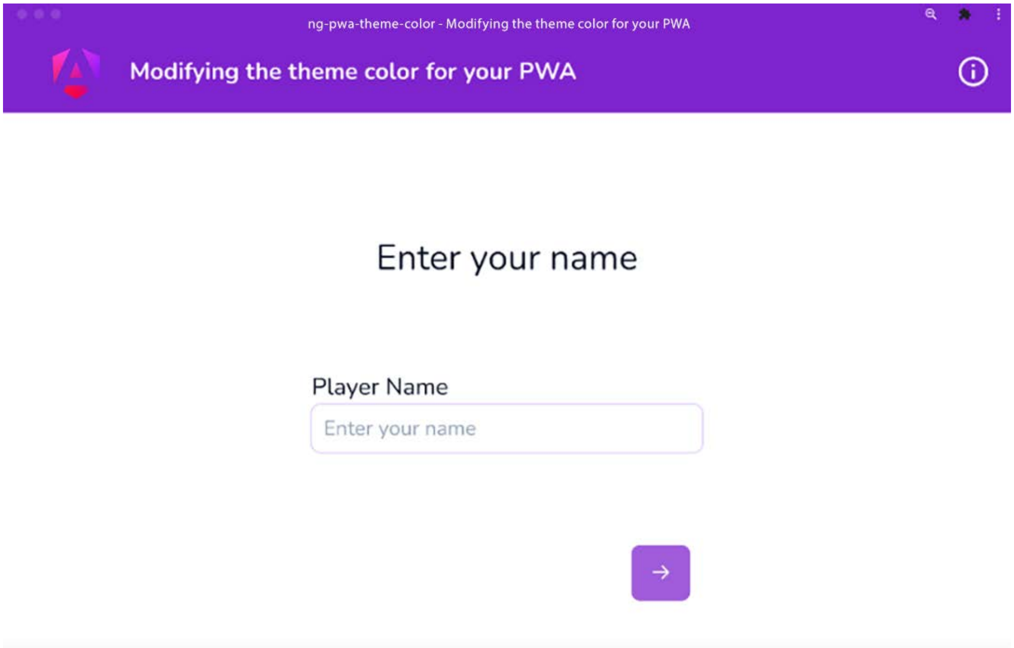

ng-pwa-theme-color - Modifying the theme color for your PWA
Modifying the theme color for your PWA
Enter your name
Player Name
Enter your name

Figure 13.13: PWA with the updated theme color

Awesomesauce! You've just learned how to update the theme color for an Angular PWA. Now that you've finished the recipe, see the next section for further reading.

See also

- Creating a PWA with the Angular CLI: `https://web.dev/creating-pwa-with-angular-cli/`

Using dark mode in your PWA

In the modern age of devices and applications, the preferences of end users have evolved a bit as well. With the increased usage of screens and devices, health is a major concern. Almost all screen devices now support dark mode. Considering this fact, if you're building a web app, you might want to provide dark mode support for it. If it is a PWA that presents itself as a native app, the responsibility is much greater. In this recipe, you'll learn how to provide dark mode for your Angular PWA. You'll learn three different ways to implement styles for dark mode, including `prefers-color-scheme` (which is the default/native CSS way) and two ways to implement it with Tailwind CSS.

Getting ready

The app that we are going to work with resides in `start/apps/chapter13/ng-pwa-dark-mode` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the cloned code repository folder, and run the following command (from the workspace's root folder) to serve the project in production mode:

   ```
   npm run serve:static chapter13 ng-pwa-dark-mode 6400
   ```

   This should open the app in a new browser tab at `https://localhost:6400`.
3. Install the application as shown in *Figure 13.8*.

   This should open the app in a native OS window, and you should see the following:

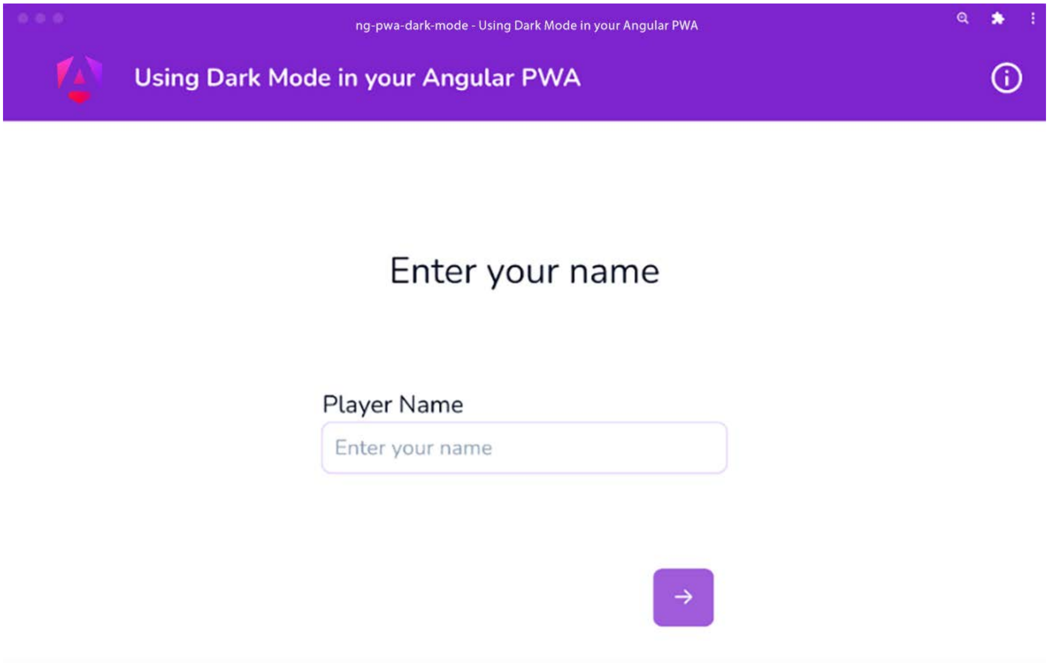
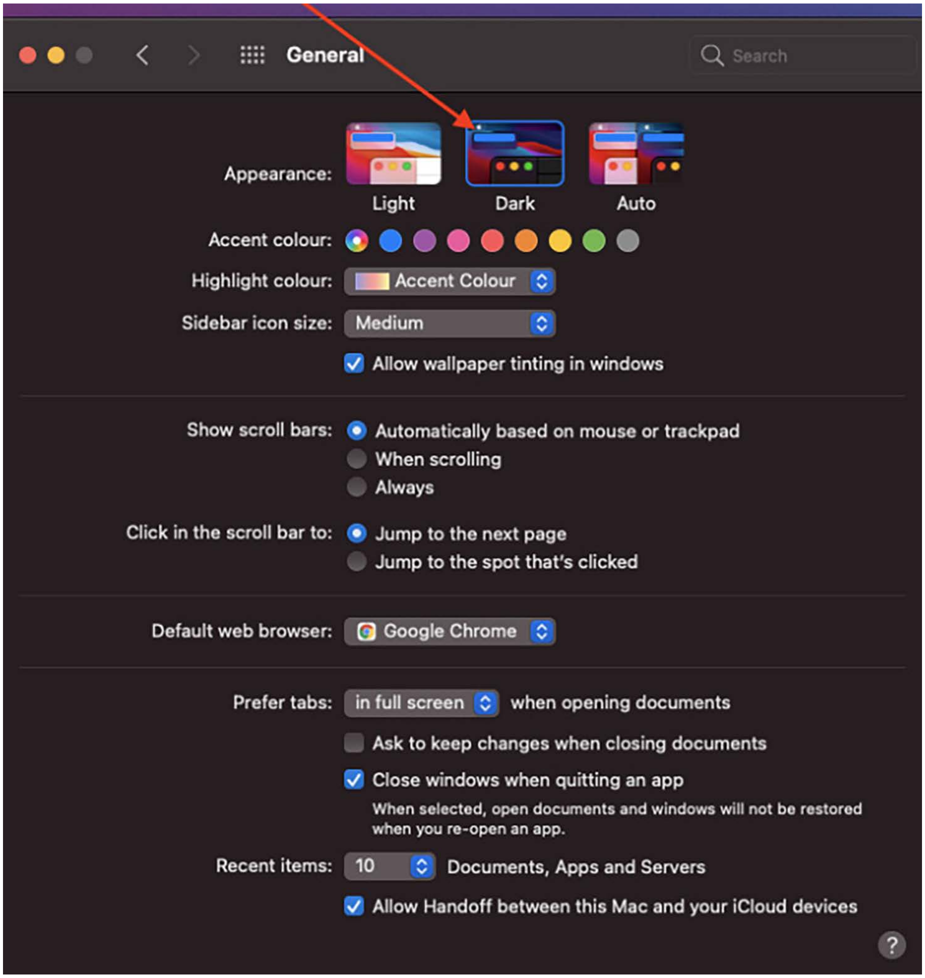

ng-pwa-dark-mode - Using Dark Mode in your Angular PWA
Using Dark Mode in your Angular PWA
Enter your name
Player Name
Enter your name

Figure 13.14: ng-pwa-dark-mode app running on http://localhost:6400

4. Now make sure you have the **Dark** theme enabled on your machine. If you're running Mac OS X, you can open **Settings** | **General** and select the **Dark** appearance, as shown in *Figure 13.15*:



Appearance: Dark

Accent colour:

Highlight colour: Accent Colour

Sidebar icon size: Medium

☑ Allow wallpaper tinting in windows

Show scroll bars:
● Automatically based on mouse or trackpad
○ When scrolling
○ Always

Click in the scroll bar to:
● Jump to the next page
○ Jump to the spot that's clicked

Default web browser: Google Chrome

Prefer tabs: in full screen when opening documents

☐ Ask to keep changes when closing documents

☑ Close windows when quitting an app

When selected, open documents and windows will not be restored when you re-open an app.

Recent items: 10 Documents, Apps and Servers

☑ Allow Handoff between this Mac and your iCloud devices

Figure 13.15: Changing the system appearance to dark mode in Mac OS X

If you run the app now, you should be able to see that the app looks the same as shown in *Figure 13.4*. However, for dark mode, we should not have a white background, etc.

Now that we have the PWA running as a native app, and the **dark** mode applied to the system, let's see the steps of the recipe in the next section.

How to do it...

As you can see, our Angular app doesn't have complete support for dark mode at the moment. That is, the UI is still too bright and looks exactly the same as what we see in light mode. We'll begin by serving the app in development mode and adding different colors for dark mode. Let's get started:

1. Serve the app in development mode by running the following command:

   ```
   npm run serve ng-pwa-dark-mode
   ```

   This should serve the app in a new browser tab at `http://localhost:4200`.

2. Now, open the `styles.scss` file to use the `prefers-color-scheme` media query. We'll use a different value for our global CSS variables to create a different view for dark mode. Update the file as follows:

   ```
   @media (prefers-color-scheme: dark) {
     html, body {
       background-color: rgb(30 41 59);
     }
   }
   ```

   If you refresh the app again in the browser tab, you'll see that the background has changed but the text is still dark and is not easily visible, as shown in *Figure 13.16*:

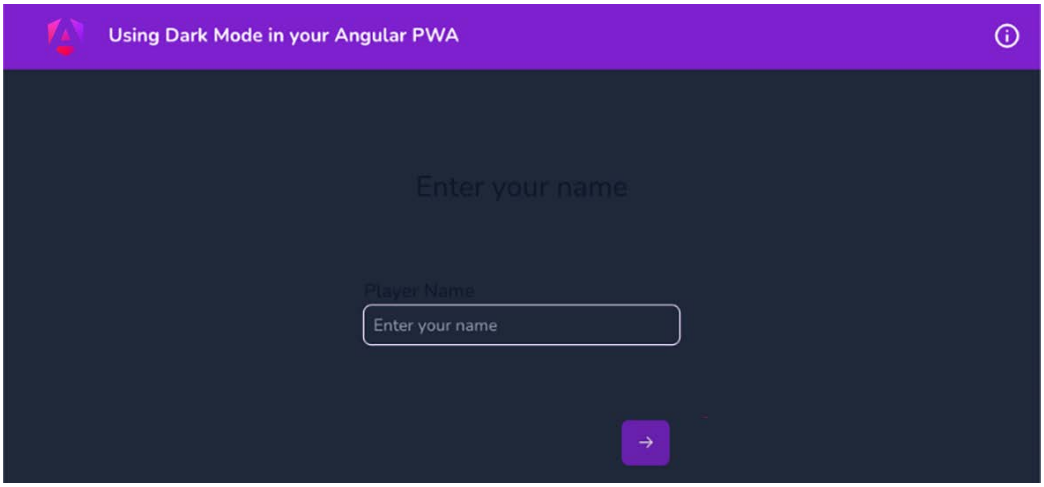
   

   Figure 13.16: Dark text and a dark background for dark mode

   To fix the styles, we need to provide the styles for dark mode individually to the necessary elements.

3. Since we are using Tailwind CSS, we are going to use the Tailwind way of applying dark mode to the input label and the input element. Update `src/app/game/game.component.html` as follows:

```
<app-game-stepper [linear]="true">
  <cdk-step...>
    ...
    <form ...>
      <div ...>
        <label for="nameInput" class="form-label dark:text-white">
  Player Name
  </label>
        <input
          type="text"
          formControlName="name"
           class="form-control dark:!text-white dark:placeholder:!text-
           white"
          id="nameInput"
          placeholder="Enter your name"
        />
        ...

</app-game-stepper>
```

Notice that for the input, we're using an exclamation sign in the `dark:!text-white` statement. This is to mark the `text-white Tailwind` CSS class as important which results in the relevant CSS style having the `!` important statement.

4. Now let's change the `game-stepper.component.html` file to update the style for the step heading as well, as follows:

```
<div class="game-stepper">
  <header>
    <h3 class="text-2xl dark:text-white" *ngIf="selected">
      ...
    </h3>
  </header>
...
```

If you refresh the application, you should be able to see the heading in white, since we're using dark mode on our device, as follows:

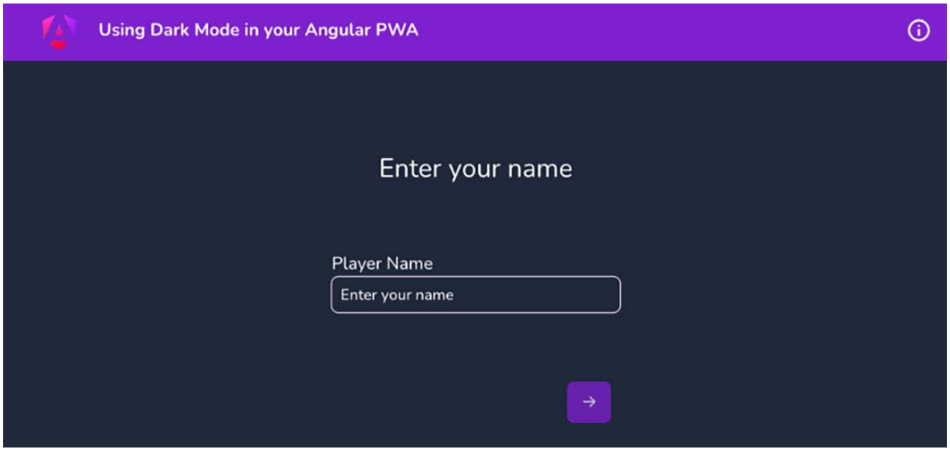

Using Dark Mode in your Angular PWA
Enter your name
Player Name
Enter your name

Figure 13.17: Heading and input colors fixed for dark mode

5. Now let's fix the numbered cards on the dice-rolling screen. You can see that in dark mode, the backgrounds on the cards are not visible, as shown in *Figure 13.18*:

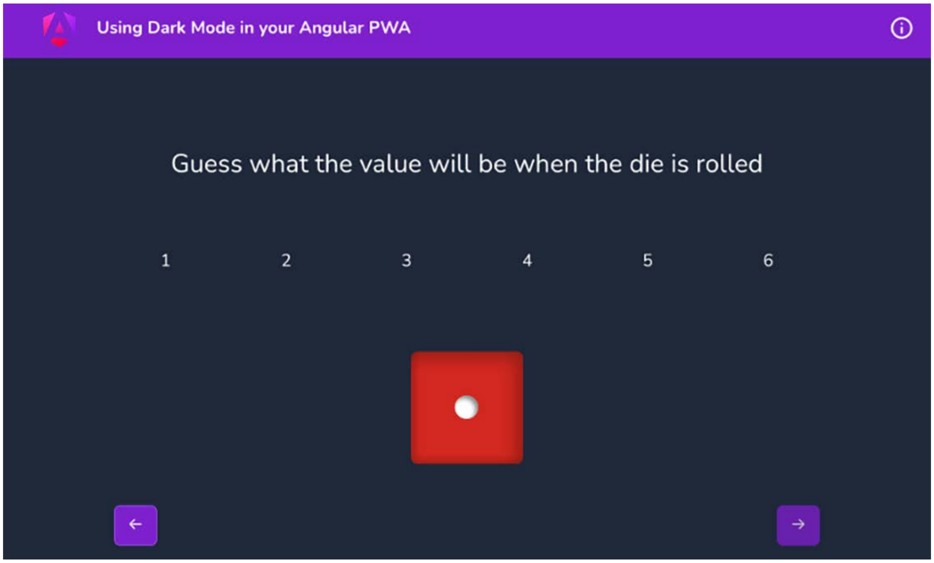
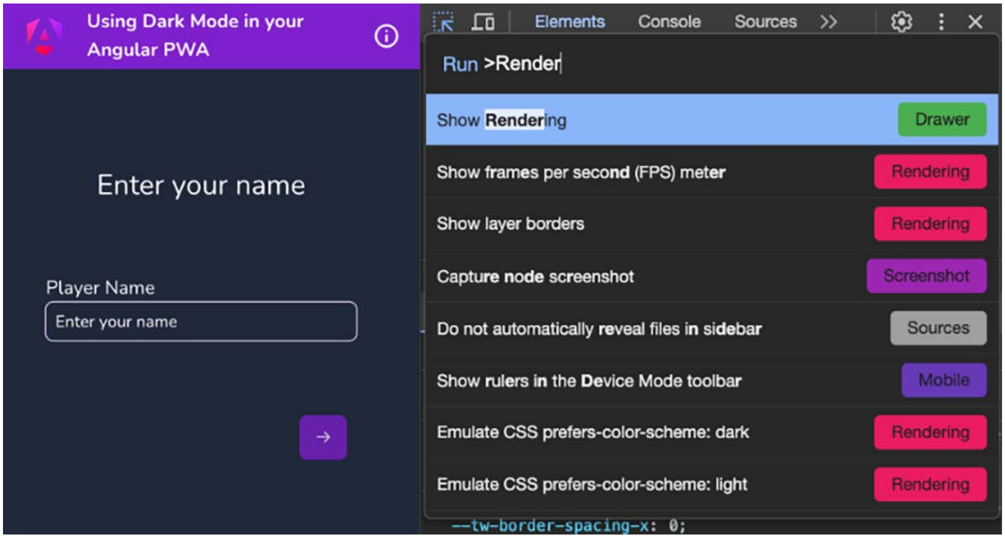

Using Dark Mode in your Angular PWA
Guess what the value will be when the die is rolled
1
2
3
4
5
6

Figure 13.18: Cards' backgrounds not visible in dark mode

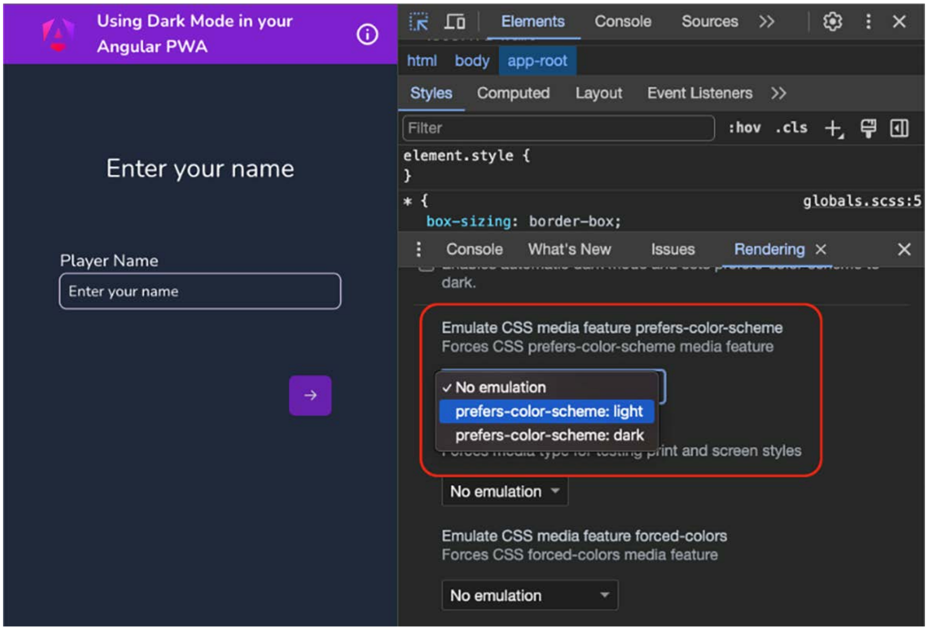
6. Let's simulate the dark and light modes using Chrome DevTools as it provides a really nice way to do so. Open Chrome DevTools and then open the **Command** menu. On macOS X, the keys are *Cmd* + *Shift* + *P*. On Windows, they are *Ctrl* + *Shift* + *P*. Then type `Render` and select the **Show Rendering** option, as we can see in *Figure 13.19*:


Using Dark Mode in your Angular PWA
Enter your name
Player Name
Enter your name
Elements
Console
Sources
Run >Render
Show Rendering
Drawer
Show frames per second (FPS) meter
Rendering
Show layer borders
Rendering
Capture node screenshot
Screenshot
Do not automatically reveal files in sidebar
Sources
Show rulers in the Device Mode toolbar
Mobile
Emulate CSS prefers-color-scheme: dark
Rendering
Emulate CSS prefers-color-scheme: light
Rendering

```
--tw-border-spacing-x: 0;
```

Figure 13.19: Open the rendering view using the Show Rendering option

7. Now, in the **Rendering** tab, toggle the **prefers-color-scheme** emulation for light and dark modes, as shown in *Figure 13.20*:


Using Dark Mode in your Angular PWA
Enter your name
Player Name
Enter your name
Elements
Console
Sources
html
body
app-root
Styles
Computed
Layout
Event Listeners
Filter
:hov
.cls
Console
What's New
Issues
Rendering
✓ No emulation
prefers-color-scheme: light
prefers-color-scheme: dark

```
element.style {
}
* {                                   globals.scss:5
  box-sizing: border-box;
```

dark.

Emulate CSS media feature prefers-color-scheme
Forces CSS prefers-color-scheme media feature

print and screen styles

No emulation

Emulate CSS media feature forced-colors
Forces CSS forced-colors media feature

No emulation

Figure 13.20: Simulate prefers-color-scheme modes

8. Now let us update the numbered cards' backgrounds. Since the styles for these are being applied from the `.scss` file, we're going to use another technique of Tailwind CSS to apply the styles in the `.scss` file. Update `value-guesser.component.scss` as follows:

```
...
.values {
  ...
  &__value-box {
    width: 5rem;
    height: 5rem;
    color: #fff;
    @apply dark:bg-slate-500;
    ...
  }
}
```

9. If you look at the dice-rolling view now, you should see the changed colors, as shown in *Figure 13.21*:

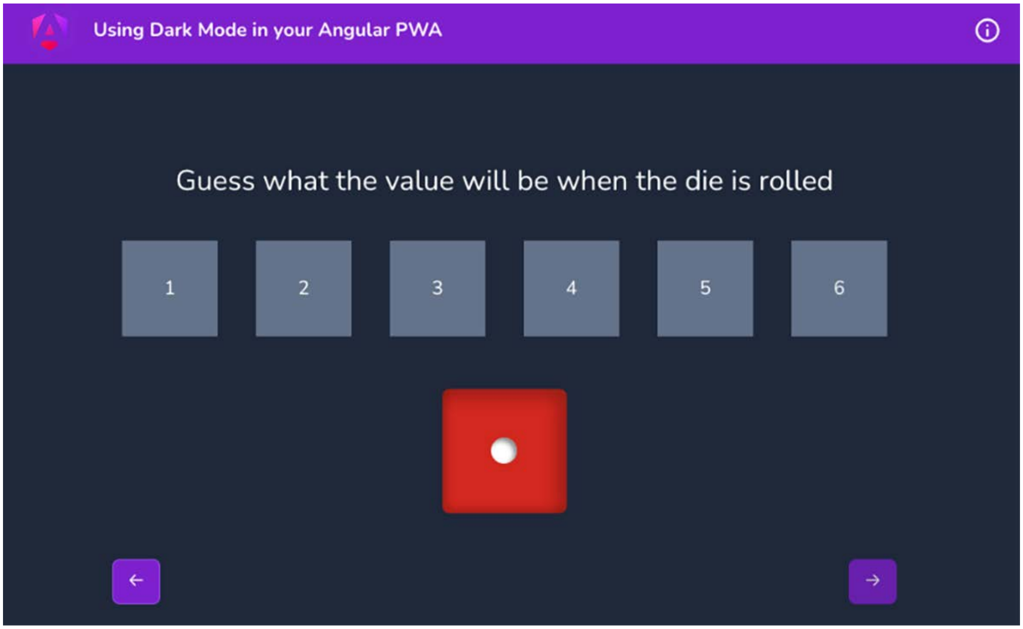

Using Dark Mode in your Angular PWA
Guess what the value will be when the die is rolled
1
2
3
4
5
6

Figure 13.21: Cards' backgrounds in dark mode

10. Now test both modes using the Chrome DevTools.
11. Uninstall the existing PWA by opening it and then selecting the **Uninstall** option from the **More** menu, as shown in *Figure 13.12*. Make sure to check the box that says **Also clear data from Chrome (...)** when prompted.

12. Run the following command to serve the production app on the browser and then navigate to `http://localhost:6400`:

    ```
    npm run serve:static chapter13 ng-pwa-dark-mode 6400
    ```

    You may need to clear your cache when this opens in the browser. You can press *Ctrl* + *Shift* + *R* on Windows and *Cmd* + *Shift* + *R* on Mac OS X to do a hard reload.

13. Wait for a few seconds for the **Install** button to show up in the address bar. Then install the PWA like in *Figure 13.8*.
14. As soon as you run the PWA now, you should see the dark mode view, as shown in *Figure 13.22*, if your system's appearance is set to dark mode:

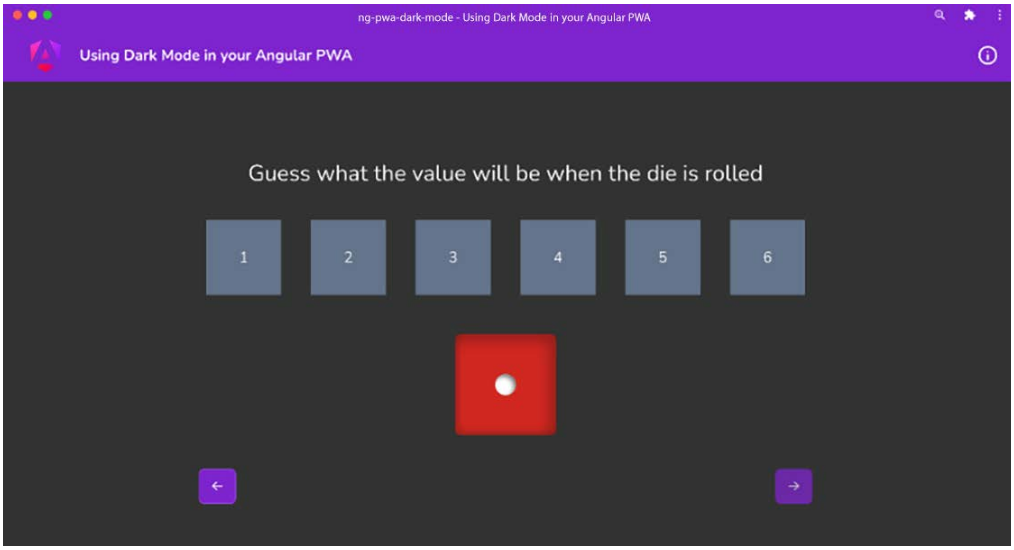

ng-pwa-dark-mode - Using Dark Mode in your Angular PWA
Using Dark Mode in your Angular PWA
Guess what the value will be when the die is rolled
1
2
3
4
5
6

Figure 13.22: Our PWA supporting dark mode out of the box

Awesome! If you switch your system appearance from dark mode to light mode or vice versa, you should see the PWA reflecting the appropriate colors. You now know three ways to implement styles for dark mode, including the native CSS way and the Tailwind CSS way. Now that you know how to support dark mode in your PWA, refer to the next section to see links for further reading.

See also

- Prefers color scheme: `https://web.dev/prefers-color-scheme/`
- Using color scheme with `prefers-color-scheme`: `https://web.dev/color-scheme/`

Providing a custom installable experience in your PWA

We know that PWAs are installable. This means they can be installed on your devices like a native application. However, when you first open the app in the browser, it totally depends on the browser as to how it shows the **Install** option; it varies from browser to browser. It also It also might not be clearly visible or clearly visible. Also, you might want to show the **Install** prompt at some special point in the app instead of the app launch, Consider displaying the Install prompt at a specific, user-friendly point within the app, rather than right at the app's launch, as this can be perceived as annoying by some users. Luckily, we have a way to provide our own custom dialog/prompt for the installation option for our PWAs. And that is what we'll learn about in this recipe.

Getting ready

The app that we are going to work with resides in `start/apps/chapter13/ng-pwa-cust-installation` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the cloned code repository folder, and run the following command (from the workspace's root folder) to serve the project in production mode:

   ```
   npm run serve:static chapter13 ng-pwa-cust-installation 7000
   ```

   This should open the app in a new browser tab at `https://localhost:7000`.

3. Install the application as shown in *Figure 13.8*.

   This should open the app in a native OS window, and you should see the following:

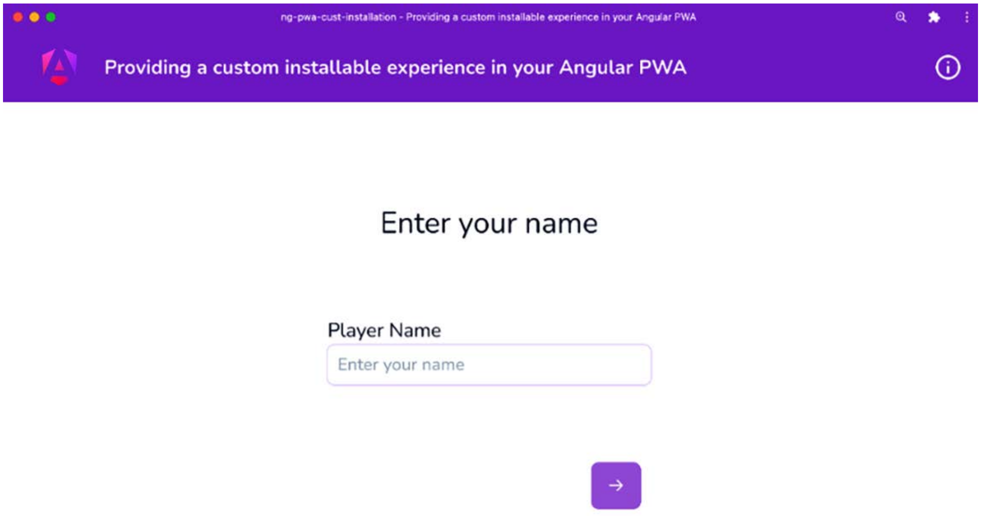

ng-pwa-cust-installation - Providing a custom installable experience in your Angular PWA
Providing a custom installable experience in your Angular PWA
Enter your name
Player Name
Enter your name

Figure 13.23: ng-pwa-cust-installation running at http://localhost:7000

Now that we have the app running, let's see the steps of the recipe in the next section.

How to do it...

We have the Dice Guesser application in which you roll the dice and guess the output. For this recipe, we'll prevent the default installation prompt and will show it only when the user has made a correct guess. Let's begin:

1. First, create a service that will show our custom installable prompt in the next steps. In the project root, run the following command:

   ```
   cd start && nx g service services/installable-prompt --project ng-pwa-
   cust-installation
   ```

   When asked, use the `@schematics/angular:service` schematics.

2. Next, open the created file, `installable-prompt.service.ts`, and update the code as follows:

   ```
   import { Injectable } from '@angular/core';

   declare global {
     interface WindowEventMap {
       beforeinstallprompt: BeforeInstallPromptEvent;
     }
   }

   @Injectable({
     providedIn: 'root'
   })
   export class InstallablePromptService {
     installPromptEvent!: BeforeInstallPromptEvent;
     constructor() {
       window.addEventListener('beforeinstallprompt',
         this.handleInstallPrompt.bind(this))
     }

     handleInstallPrompt(ev: BeforeInstallPromptEvent) {
       ev.preventDefault();
       this.installPromptEvent = ev;
       console.log('before install prompt event fired', ev);
       window.removeEventListener('beforeinstallprompt',
         this.handleInstallPrompt)
     }
   }
   ```

   You will notice that TypeScript isn't happy with our code.

3. In this step, we need to define the `BeforeInstallPrompt` event type. Let's create a folder named `types` inside the `src` folder. Create a file named `installation-prompt.d.ts` and add the following code to it:

```
interface BeforeInstallPromptEvent extends Event {
  readonly platforms: string[];
  readonly userChoice: Promise<{
    outcome: "accepted" | "dismissed";
    platform: string;
  }>;
  prompt(): Promise<void>;
}
```

4. We must add the `types` folder in `tsconfig.json` so we can load the types. Update the `tsconfig.json` file as follows:

```
{
  "compilerOptions": {
    ...,
    "noFallthroughCasesInSwitch": true,
    "typeRoots": [
      "./src/types"
    ]
  },
  ...
}
```

5. Next, let's build the custom installation banner we'll show to the user to trigger the installation prompt. Update `app.component.html` and add the following code AFTER the `<main>`, tag as shown here:

```
...
<main class="content" role="main">
  <router-outlet></router-outlet>
</main>
<div id="installPrompt" class="fixed inset-x-0 bottom-10
  p-4">
  <div class="rounded-lg bg-purple-600 px-4 py-3 text-
    white shadow-lg">
    <p class="text-center text-sm font-medium">
      Love the Dice Game?
      <a href="#" class="inline-block underline">
        Install it!
      </a>
      </p>
    </div>
  </div>
```

6. Now run the application again using the following command to see the installation prompt, as shown in *Figure 13.24*:

```
npm run serve:static chapter13 ng-pwa-cust-installation 7000
```

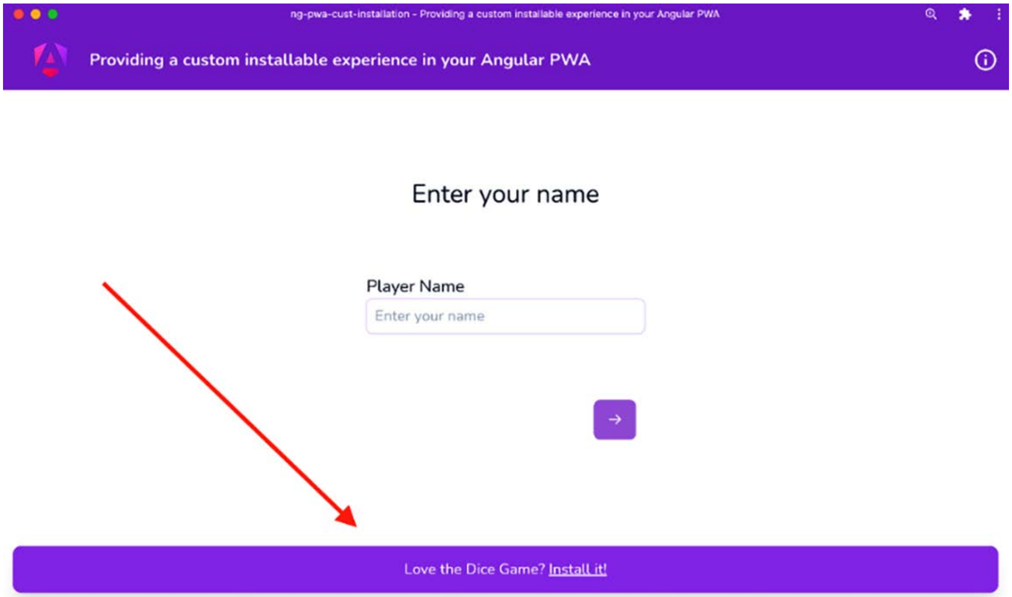

ng-pwa-cust-installation - Providing a custom installable experience in your Angular PWA
Providing a custom installable experience in your Angular PWA
Enter your name
Player Name
Enter your name
Love the Dice Game? Install it!

Figure 13.24: Custom installation prompt being displayed in the PWA

Sometimes you will see cached versions of the application due to running multiple PWAs on the same port, for instance. In this case, open the Chrome DevTools, go to the **Application** tab, click the **Service Workers** option from the left menu, and check the **Bypass for network** option.

7. We're now going to create a `signal` inside the `InstallablePromptService` class, based on which we'll show or hide the custom installation banner. Update the `installable-prompt.service.ts` file as follows:

```
import { Injectable, signal } from '@angular/core';
...
export class InstallablePromptService {
  installPromptEvent!: BeforeInstallPromptEvent;
  isPromptBannerShown = signal(false);
  constructor() {
    ...
  }
}
```

8. Let's import the `InstallationPromptService` class in the app component so we can use it in the template for the custom installation banner. Update the `app.component.ts` file as follows:

```
...
import { Component, inject } from '@angular/core';
...
import { InstallablePromptService } from './services/installable-prompt.
service';

...
export class AppComponent {
  promptService = inject(InstallablePromptService);
}
```

9. Update the template now to use the `signal` from the injected `InstallablePromptService` in the app component. Update the `app.component.html` file as follows:

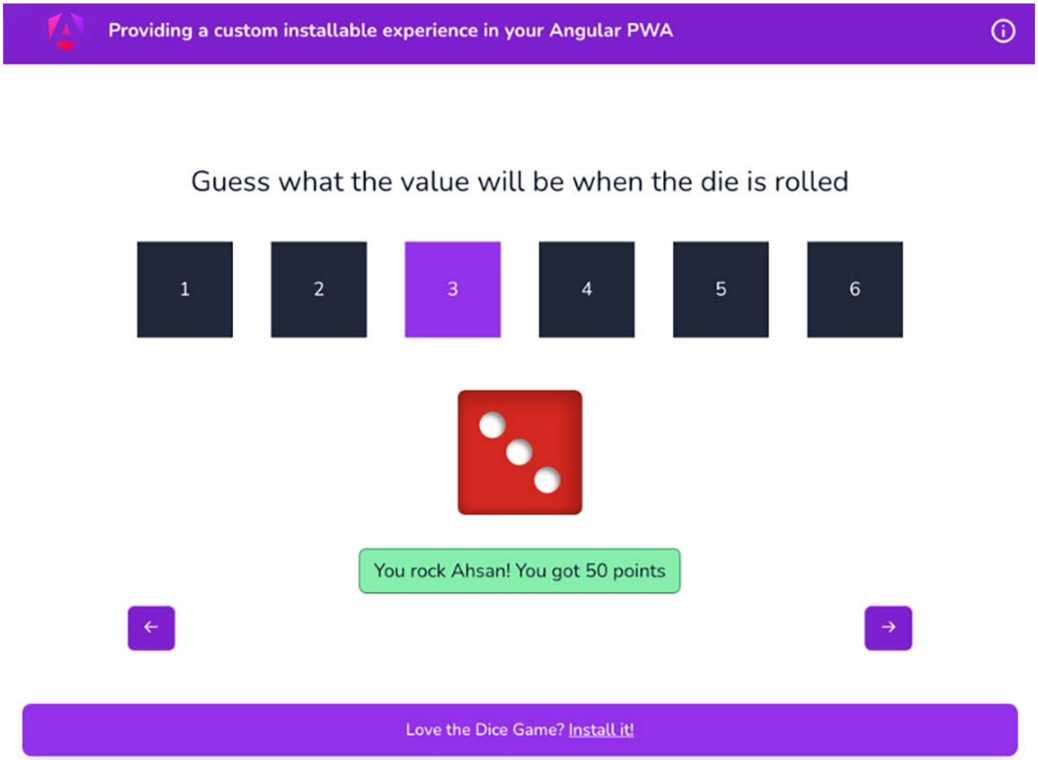
```
...

<main class="content" role="main">
  <router-outlet></router-outlet>
</main>
 @if(promptService.isPromptBannerShown()) {
  <div id="installPrompt" class="fixed inset-x-0
     bottom-10 p-4">
    ...
  </div>
}
```

Restart the server by running `npm run serve:static chapter13 ng-pwa-cust-installation 7000` and you'll notice that the custom installation banner doesn't show anymore. That's because now we have to flip the `signal` in the `InstallationPromptService` class when the user guesses the score correctly.

10. Uninstall the existing PWA by opening it and then selecting the **Uninstall** option from the **More** menu, as shown in *Figure 13.12*. Make sure to check the box that says **Also clear data from Chrome (...)** when prompted.
11. Update the `src/app/game/game.component.ts` file to flip the `signal`, as follows:

```
import { InstallablePromptService } from '../services/installable-prompt.
service';
...
export class GameComponent {
  ...
  leaderboardService = inject(LeaderboardService);
  promptService = inject(InstallablePromptService);
  ...

  showResult(diceSide: IDiceSide) {
    ...
    if (!this.isCorrectGuess) {
      return;
    }
    if (this.promptService.installPromptEvent) {
      this.promptService.isPromptBannerShown.set(true);
    }
    ...
  }
}
```

If you now run the app again, as shown in *step 6*, and guess the dice rolls, and you guess the correct one, you should be able to see the custom installation banner, as shown in *Figure 13.25*:



Providing a custom installable experience in your Angular PWA

Guess what the value will be when the die is rolled

You rock Ahsan! You got 50 points

Love the Dice Game? Install it!

Figure 13.25: Custom installation banner being displayed after guessing

12. Now we'll implement what happens when we click the **Install it!** link from the custom installation banner. Create a new method in the file `installable-prompt.service.ts` as follows:

```
...
export class InstallablePromptService {
  ...

  handleInstallPrompt(ev: BeforeInstallPromptEvent) {...}

  async showInstallPrompt() {
    if (!this.installPromptEvent) {
      return;
    }
    await this.installPromptEvent.prompt();
    const { outcome } =
       await this.installPromptEvent.userChoice;
    console.log('The choice of user is ', outcome);
    this.isPromptBannerShown.set(false);
  }
}
```

13. We will now call this method from the custom installation banner to show the browser's installation prompt. Update the `app.component.html` file, as follows:

```
...
<main class="content" role="main">
  <router-outlet></router-outlet>
</main>
<div id="installPrompt" *ngIf=
  "promptService.isPromptBannerShown()"
  class="fixed inset-x-0 bottom-4 p-4">
  <div class="rounded-lg bg-purple-600
    px-4 py-3 text-white shadow-lg">
    <p class="text-center text-sm font-medium">
      Love the Dice Game?
      <a (click)="promptService.showInstallPrompt()"
        href="#" class="inline-block underline">
        Install it!
      </a>
    </p>
  </div>
</div>
```

Now, if you rerun the app in the browser and guess the dice's next value, you can click on the **Install it!** link to see the browser prompt, as shown in *Figure 13.26*:

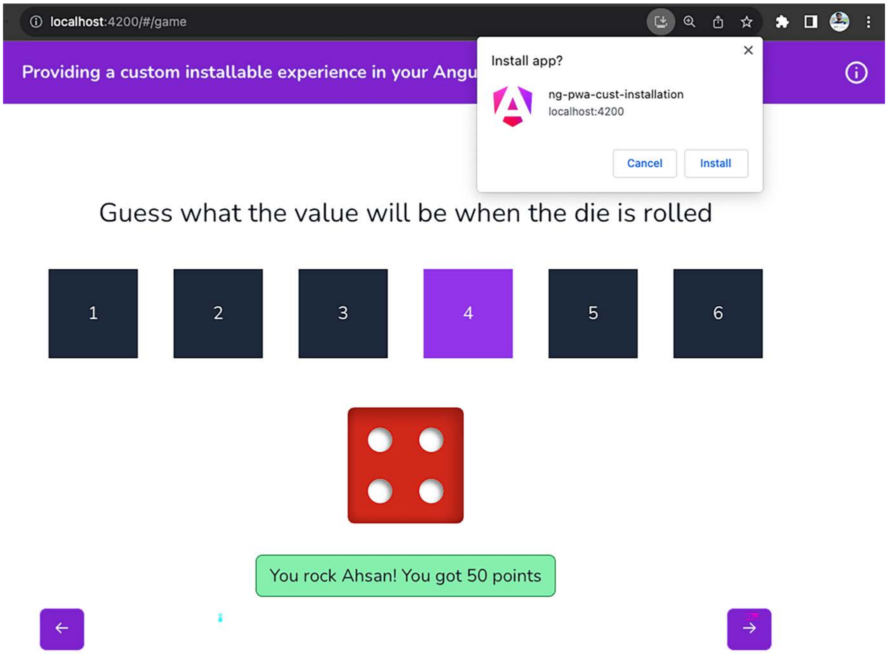

localhost:4200/#/game
Providing a custom installable experience in your Angu
Install app?
ng-pwa-cust-installation
localhost:4200
Cancel
Install
Guess what the value will be when the die is rolled
1
2
3
4
5
6
You rock Ahsan! You got 50 points

Figure 13.26: Browser installation prompt being displayed

14. Finally, we don't want to show the custom installation banner when the app is running as a PWA. And right now, it does. To fix that, we'll add some styles to `app.component.scss`, as follows:

```
@media all and (display-mode: standalone) {
  #installPrompt {
    display: none !important;
  }
}
```

If you install the PWA now and guess the dice's next value correctly, you won't see the custom installation banner anymore.

Awesome! You can now play around with the app by installing and uninstalling the PWA a few times and trying out all the combinations of the user choosing to install or not to install the app. It's all fun and games. Now that you know how to implement a custom installation banner for an Angular PWA, move on to the next section to understand how it works.

How it works...

The heart of this recipe is the `beforeinstallprompt` event. It is a standard browser event that is supported in the latest versions of Chrome, Firefox, Safari, Opera, UC Browser (Android version), and Samsung Internet, that is, almost all major browsers. The event has a `prompt` method, which shows the browser's default prompt on the device. In the recipe, we create `InstallablePromptService` and store the event in a property called `installPromptEvent`. This is so we can use it later on demand when the user has guessed a correct dice-roll value. Note that as soon as we receive the `beforeinstallprompt` event, we remove the event listener from the `window` object, so we only save the event once. That is when the app starts. If the user chooses not to install the app, we don't show the prompt again within the same session. However, if the user refreshes the app, they will still get the prompt one time for the first correct guess. We could go one step further to save this state in `localStorage` to avoid showing the prompt after the page refreshes, but that's not part of this recipe.

For the custom installation banner, we use a template based on Tailwind CSS. Note that on the custom installation banner, we have a link. When we click this link, the `showInstallPrompt` method from the `InstallablePromptService` class is called. In that method, we use the event, that is, the `this.installPromptEvent` property's `prompt` method to show the browser's prompt. Note that after we've displayed the browser's prompt, we set the signal `isPromptBannerShown` value to `false`, so it hides the custom install banner. This is also so we don't show the prompt again in the same session until the user refreshes the page.

Finally, we also use some CSS to not show the custom installation banner at all if the app is launched as a PWA. This is important because if it is already a PWA, it doesn't make sense to show an install prompt. Therefore, we use an `@media` query with `display-mode: standalone`, which checks if the app is running as a PWA. In that CSS rule, we hide the custom installation banner.

Now that you understand how it all works, refer to the next section to see links for further reading.

See also

- How to provide your own app-install experience (`web.dev`): `https://web.dev/customize-install/`

Precaching requests using an Angular service worker

With the addition of service workers in our previous recipes, we've seen that they already cache the assets and serve them using the service worker if we go into Offline mode. But what about network requests? If the user goes offline and refreshes the application right now, the network requests fail because they're not cached with the service worker. This results in a broken offline user experience. In this recipe, we'll configure the service worker to precache network requests, so the app works fluently in Offline mode as well.

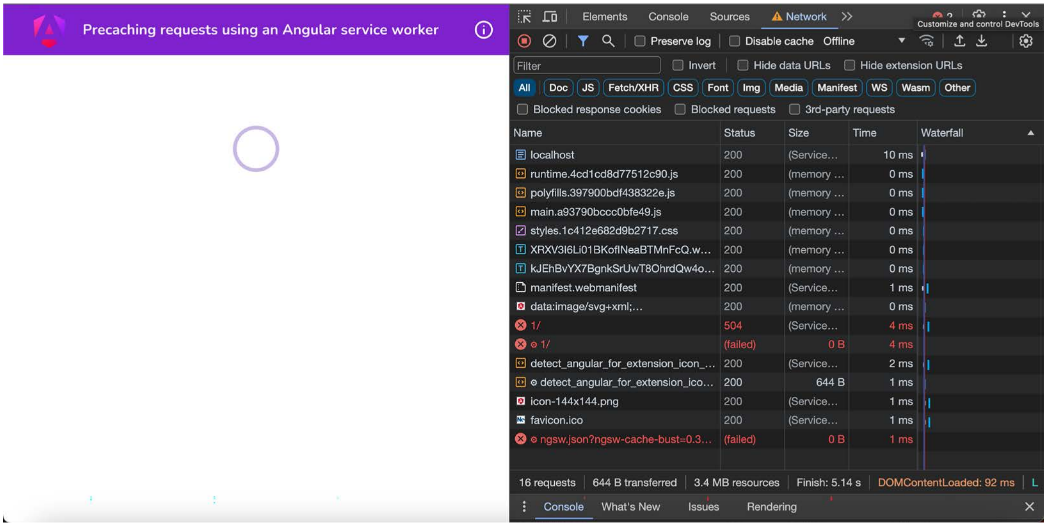
Getting ready

The app that we are going to work with resides in `start/apps/chapter13/ng-pwa-precaching` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the cloned code repository folder, and run the following command (from the workspace's root folder) to serve the project in production mode:

   ```
   npm run serve:static chapter13 ng-pwa-precaching 7600
   ```

   This should open the app in a new browser tab at `https://localhost:7600`.
3. Refresh the page once.
4. Now, switch to Offline mode as shown in *Figure 13.2*. If you go to the **Network** tab and filter the requests using the query results, you should see that the requests fail, as shown in *Figure 13.27*:


Precaching requests using an Angular service worker
Elements
Console
Sources
Network
Customize and control DevTools
Preserve log
Disable cache
Filter
Invert
Hide data URLs
Hide extension URLs
All
Doc
JS
Fetch/XHR
CSS
Font
Img
Media
Manifest
WS
Wasm
Other
Blocked response cookies
Blocked requests
3rd-party requests
Console
What's New
Issues
Rendering

Offline

Name	Status	Size	Time	Waterfall
localhost	200	(Service…	10 ms	
runtime.4cd1cd8d77512c90.js	200	(memory …	0 ms	
polyfills.397900bdf438322e.js	200	(memory …	0 ms	
main.a93790bccc0bfe49.js	200	(memory …	0 ms	
styles.1c412e682d9b2717.css	200	(memory …	0 ms	
XRXV3I6Li01BKoflNeaBTMnFcQ.w…	200	(memory …	0 ms	
kJEhBvYX7BgnkSrUwT8OhrdQw4o…	200	(memory …	0 ms	
manifest.webmanifest	200	(Service…	1 ms	
data:image/svg+xml;…	200	(memory …	0 ms	
1/	504	(Service…	4 ms	
⚙ 1/	(failed)	0 B	4 ms	
detect_angular_for_extension_icon_…	200	(Service…	2 ms	
⚙ detect_angular_for_extension_ico…	200	644 B	1 ms	
icon-144x144.png	200	(Service…	1 ms	
favicon.ico	200	(Service…	1 ms	
⚙ ngsw.json?ngsw-cache-bust=0.3…	(failed)	0 B	1 ms	

16 requests | 644 B transferred | 3.4 MB resources | Finish: 5.14 s | DOMContentLoaded: 92 ms | L

Figure 13.27: Offline experience broken due to not caching the network requests

Now that we see the network requests failing, let us see the steps of the recipe to fix this in the next section.

How to do it...

As you can see in *Figure 13.27*, the app still loads. We can see the header and the loader. However, the API (fetch) calls do not work since we are simulating an offline network situation using Chrome DevTools. This is because the service worker is not configured yet to cache the data request. Let us get started with the recipe to fix this:

1. To cache the network requests, open the `ngsw-config.json` file inside the `start/apps/chapter13/ng-pwa-precaching` folder and update it as follows:

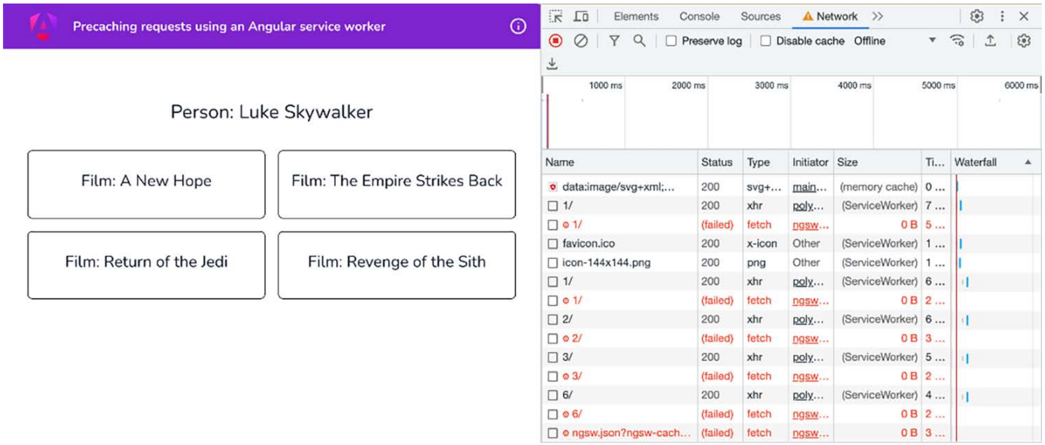
   ```
   {
     "$schema": "../../../node_modules/@angular/service-worker/config/
   schema.json",
     "index": "/index.html",
     "assetGroups": [
       ...
   ```

```
  ],
  "dataGroups": [
    {
      "name": "swapi.dev",
      "urls": [
        "https://swapi.dev/api/*"
      ],
      "cacheConfig": {
        "strategy": "freshness",
        "maxSize": 100,
        "maxAge": "2d"
      }
    }
  ]
}
```

2. Let us test the application now. Build and run the app in production mode using the following command:

```
npm run serve:static chapter13 ng-pwa-precaching 7600
```

3. Now navigate to `http://localhost:7600`. Make sure **Network throttling** is not enabled, that is, you are not in **Offline** mode.
4. Clear the app data using Chrome DevTools using the **Application** tab | the **Storage** panel. Or press *Cmd* + *Shift* + *R* (macOS X) or *Ctrl* + *Shift* + *R* (Windows) to do a hard reload.
5. Refresh the app once more to let the service worker cache the API requests.
6. In Chrome DevTools, go to the **Network** tab and switch to Offline mode, as shown in *Figure 13.2*. Refresh the app again. You should see the data from Swapi despite being offline. The network calls are served from the service worker, as shown in *Figure 13.28*:


Precaching requests using an Angular service worker
Person: Luke Skywalker
Film: A New Hope
Film: The Empire Strikes Back
Film: Return of the Jedi
Film: Revenge of the Sith
Elements
Console
Sources
Network
☐ Preserve log
☐ Disable cache
Offline
1000 ms
2000 ms
3000 ms
4000 ms
5000 ms
6000 ms

Name	Status	Type	Initiator	Size	Ti...	Waterfall
data:image/svg+xml;...	200	svg+...	main...	(memory cache)	0 ...	
☐ 1/	200	xhr	poly...	(ServiceWorker)	7 ...	
☐ 1/	(failed)	fetch	ngsw...	0 B	5 ...	
☐ favicon.ico	200	x-icon	Other	(ServiceWorker)	1 ...	
☐ icon-144x144.png	200	png	Other	(ServiceWorker)	1 ...	
☐ 1/	200	xhr	poly...	(ServiceWorker)	6 ...	
☐ 1/	(failed)	fetch	ngsw...	0 B	2 ...	
☐ 2/	200	xhr	poly...	(ServiceWorker)	6 ...	
☐ 2/	(failed)	fetch	ngsw...	0 B	3 ...	
☐ 3/	200	xhr	poly...	(ServiceWorker)	5 ...	
☐ 3/	(failed)	fetch	ngsw...	0 B	2 ...	
☐ 6/	200	xhr	poly...	(ServiceWorker)	4 ...	
☐ 6/	(failed)	fetch	ngsw...	0 B	2 ...	
☐ ngsw.json?ngsw-cach...	(failed)	fetch	ngsw...	0 B	3 ...	

Figure 13.28: Network calls being cached by the service worker

And boom! You have just learned how to configure a service worker in an Angular app to cache network/data requests. And you can install the PWA and use it even if you are offline. Awesome, right?

Now that we have finished the recipe, let's see how it all works in the next section.

How it works...

The heart of this recipe is the `ngsw-config.json` file. This file is used by the `@angular/service-worker` package when generating the service worker file. The file already contains a JSON object out of the box when we use the `@angular/pwa` schematics by running `ng add @angular/pwa`. This JSON contains a property called `assetGroups`, which configures the caching of the assets based on the provided configuration. For this recipe, we wanted to cache network requests along with the assets. Therefore, we added the new property `dataGroups` in the JSON object. Let us have a look at the configuration:

```
    "dataGroups": [
      {
        "name": "swapi.dev",
        "urls": [
          "https://swapi.dev/api/*"
        ],
        "cacheConfig": {
          "strategy": "freshness",
          "maxSize": 100,
          "maxAge": "2d"
        }
      }
    ]
```

As you can see, `dataGroups` is an array. We can provide different configuration objects as elements to it. Each configuration has a `name`, an array of `urls`, and a `cacheConfig` that defines the caching strategy. For our configuration, we use a wildcard with the API URL, that is, we use `urls:` ["`https://swapi.dev/api/*`"]. For the `cacheConfig`, we are using the `freshness` strategy, which means the app will always fetch the data from its origin first. If the network is unavailable, then it will use the response from the service worker cache. An alternate strategy is `performance`, which first looks up the service worker for a cached response. If there is nothing in the cache for the URL (or URLs), then it fetches the data from the actual origin. The `maxSize` property defines how many requests can be cached for the same pattern (or set of URLs). The `maxAge` property defines how long the cached data would live in the service worker cache.

Now that you know how the recipe works, see the next section for links to further reading.

See also

- Angular service worker intro: `https://angular.io/guide/service-worker-intro`
- Angular service worker config: `https://angular.io/guide/service-worker-config`
- Creating an offline fallback page (`web.dev`): `https://web.dev/offline-fallback-page/`

Creating an App Shell for your PWA

When it comes to building fast user experiences for web apps, one of the major challenges is minimizing the critical rendering path. This includes loading the most critical resources for the target page, parsing, executing JavaScript, and so on. With an App Shell, we can render a page, or a portion of the app, at build time rather than runtime. This means the user will see the minimal pre-rendered content initially, until JavaScript and Angular kick in. This means the browser does not have to work and wait a while for the first meaningful paint. This not only provides a good user experience but also helps rank the website higher in search engines, i.e., results in better SEO. In this recipe, you will create an App Shell for an Angular PWA.

Getting ready

The app that we are going to work with resides in `start/apps/chapter13/ng-pwa-app-shell` inside the cloned repository:

1. Open the code repository in your code editor.
2. Open the terminal, navigate to the cloned code repository folder, and run the following command (from the workspace's root folder) to serve the project in production mode:

```
npm run serve ng-pwa-app-shell
```

This should open the app in a new browser tab at `http://localhost:4200`, and you should see the following:

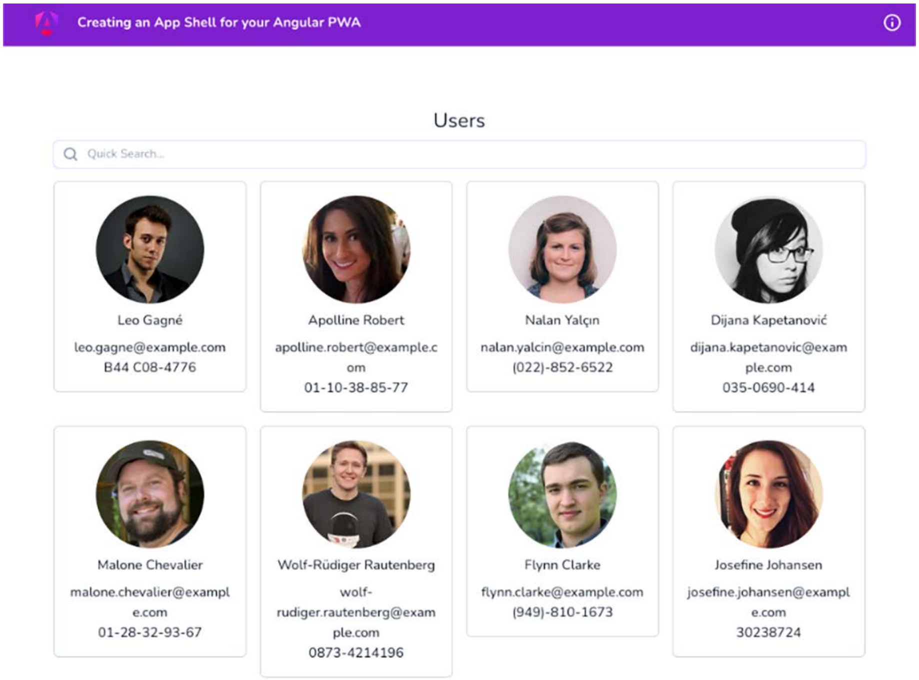
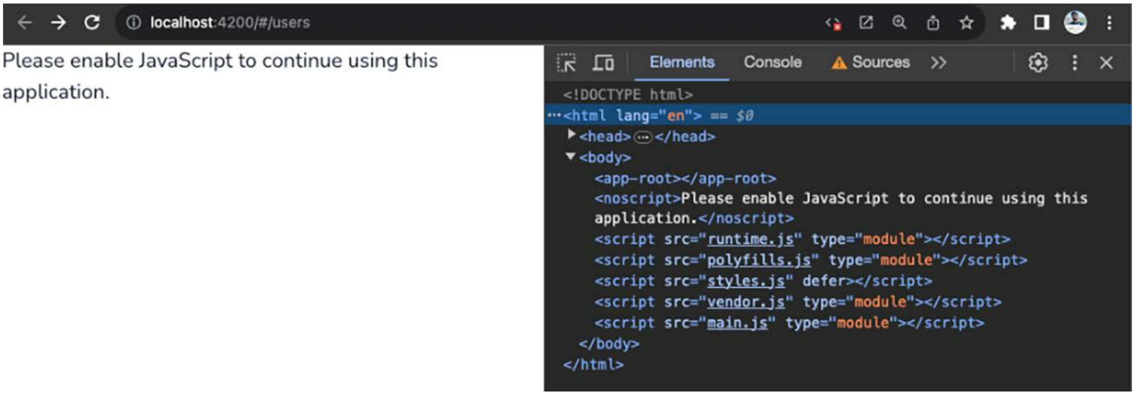

Creating an App Shell for your Angular PWA
Users
Quick Search...
Leo Gagné
leo.gagne@example.com
B44 C08-4776
Apolline Robert
apolline.robert@example.com
01-10-38-85-77
Nalan Yalçın
nalan.yalcin@example.com
(022)-852-6522
Dijana Kapetanović
dijana.kapetanovic@example.com
035-0690-414
Malone Chevalier
malone.chevalier@example.com
01-28-32-93-67
Wolf-Rüdiger Rautenberg
wolf-rudiger.rautenberg@example.com
0873-4214196
Flynn Clarke
flynn.clarke@example.com
(949)-810-1673
Josefine Johansen
josefine.johansen@example.com
30238724

Figure 13.29: ng-pwa-app-shell running on http://localhost:4200

Now we'll disable JavaScript to simulate taking a lot of time to parse JavaScript, or to simulate that there's no App Shell in place yet. Now open Chrome DevTools. Then open the command panel. The shortcut for opening the command panel is *Cmd* + *Shift* + *P* on Mac OS X and *Ctrl* + *Shift* + *P* on Windows. Type **Disable JavaScript**, select the option, and hit *Enter*. Refresh the page while Chrome DevTools is still open, and you should see a message as follows:



Please enable JavaScript to continue using this application.

```
<!DOCTYPE html>
<html lang="en"> == $0
  <head>…</head>
  <body>
    <app-root></app-root>
    <noscript>Please enable JavaScript to continue using this
    application.</noscript>
    <script src="runtime.js" type="module"></script>
    <script src="polyfills.js" type="module"></script>
    <script src="styles.js" defer></script>
    <script src="vendor.js" type="module"></script>
    <script src="main.js" type="module"></script>
  </body>
</html>
```

Figure 13.30: No App Shell present in the app

Now that we have checked for the absence of the App Shell, let's see the steps of the recipe in the next section.

How to do it...

We have an Angular application that fetches some users from an API. We will create an App Shell for this app so it can provide the first meaningful paint faster as a PWA. Let's get started:

1. First, create the App Shell for the app by running the following command from the project root:

   ```
   cd start && nx g app-shell --project=ng-pwa-app-shell
   ```

   This adds some new files to our project but also updates `project.json`. Since we're using a unique folder structure for the recipes, you need to update the `project.json` file by replacing all the `dist/ng-pwa-app-shell` instances with `dist/apps/chapter13/ng-pwa-app-shell`.

2. Update the `src/app/app-shell/app-shell.component.ts` file to import the `UsersComponent` class so we can render the `users` page in the App Shell. The code should look as follows:

   ```
   ...
   import { UsersComponent } from '../users/users.component';

   @Component({
     selector: 'app-app-shell',
   ```

```
    standalone: true,
    imports: [CommonModule, UsersComponent],
    templateUrl: './app-shell.component.html',
    styleUrls: ['./app-shell.component.css'],
  })
  ...
```

3. Now open the `app-shell.component.html` file and use the `<app-users>` element so we render the whole `UsersComponent` in the App Shell. The code should look as follows:

```
<app-users></app-users>
```

4. Finally, update `users.component.ts` to show the skeleton loader when the App Shell is being generated. Update the file as follows:

```
import { Component, inject, OnInit, PLATFORM_ID } from '@angular/core';
import { CommonModule, isPlatformBrowser } from '@angular/common';
...
export class UsersComponent implements OnInit {
 ...
  platformId = inject(PLATFORM_ID);
  isSearching = true;
  ngOnInit() {
    this.componentAlive = true;
   ...
    this.searchForm.controls['username'].valueChanges
      .pipe(
        startWith(''),
        tap(() => {
          this.isSearching = true;
        }),
        takeWhile(() => !!this.componentAlive &&
          isPlatformBrowser(this.platformId)),
        mergeMap((query) => this.userService
          .searchUsers(query))
      )
      .subscribe((users) => {
        this.users = users;
        this.isSearching = false;
      });
  }
}
```

5. Now that we have the code written for the App Shell, let's create it. Run the following command from the `workspace root` to generate the App Shell in `production` mode:

```
cd start && nx run ng-pwa-app-shell:app-shell:production
```

6. Once the App Shell is generated in *step 5*, run the following command from the `workspace root` (not from the `start` folder) to serve it using the `http-server` package:

```
npx http-server ./start/dist/apps/chapter13/ng-pwa-app-shell/browser -o
-p 4200
```

7. Make sure that JavaScript is still turned off for the app. If not, open Chrome DevTools and press *Cmd* + *Shift* + *P* for Mac OS X to open the command panel (*Ctrl* + *Shift* + *P* on Windows). Then type `Disable Javascript` and hit *Enter*, selecting the option as shown in *Figure 13.31*:

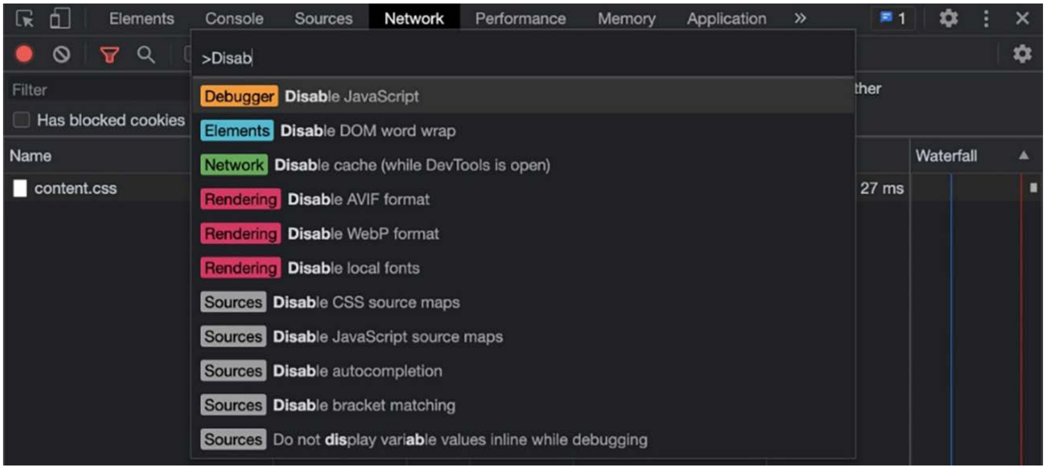
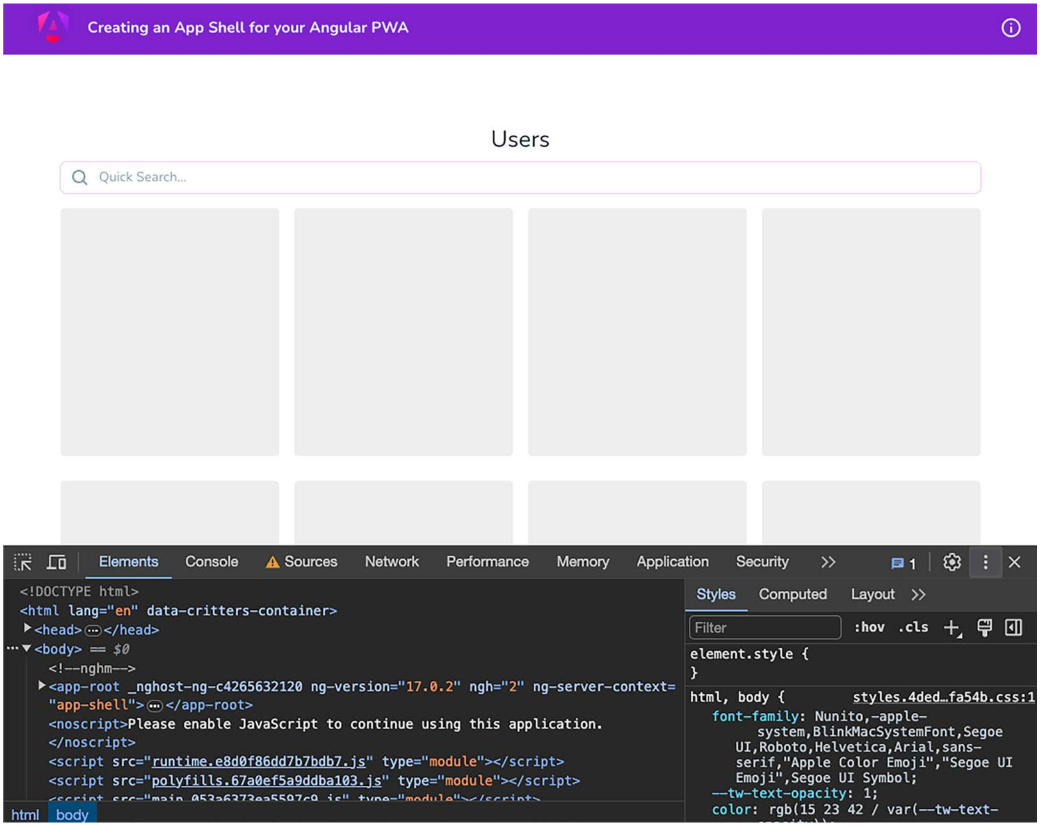

Elements
Console
Sources
Network
Performance
Memory
Application
»
1
>Disab
Filter
Has blocked cookies
ther
Name
Waterfall
content.css
27 ms
Debugger Disable JavaScript
Elements Disable DOM word wrap
Network Disable cache (while DevTools is open)
Rendering Disable AVIF format
Rendering Disable WebP format
Rendering Disable local fonts
Sources Disable CSS source maps
Sources Disable JavaScript source maps
Sources Disable autocompletion
Sources Disable bracket matching
Sources Do not display variable values inline while debugging

Figure 13.31: Disable JavaScript using Chrome DevTools

8. Refresh the app while JavaScript is disabled. You should now see the app still showing the pre-rendered users page with the skeleton loaders, despite JavaScript being disabled, as shown in *Figure 13.32*. Woohoo!


Creating an App Shell for your Angular PWA
Users
Quick Search...
Elements
Console
Sources
Network
Performance
Memory
Application
Security
1
Styles
Computed
Layout
Filter
:hov
.cls
html
body

```
<!DOCTYPE html>
<html lang="en" data-critters-container>
 ▶<head>⋯</head>
⋯▼<body> == $0
    <!--nghm-->
   ▶<app-root _nghost-ng-c4265632120 ng-version="17.0.2" ngh="2" ng-server-context=
    "app-shell">⋯</app-root>
    <noscript>Please enable JavaScript to continue using this application.
    </noscript>
    <script src="runtime.e8d0f86dd7b7bdb7.js" type="module"></script>
    <script src="polyfills.67a0ef5a9ddba103.js" type="module"></script>
```

```
element.style {
}
html, body {          styles.4ded…fa54b.css:1
   font-family: Nunito,-apple-
         system,BlinkMacSystemFont,Segoe
      UI,Roboto,Helvetica,Arial,sans-
      serif,"Apple Color Emoji","Segoe UI
      Emoji",Segoe UI Symbol;
   --tw-text-opacity: 1;
   color: rgb(15 23 42 / var(--tw-text-
```

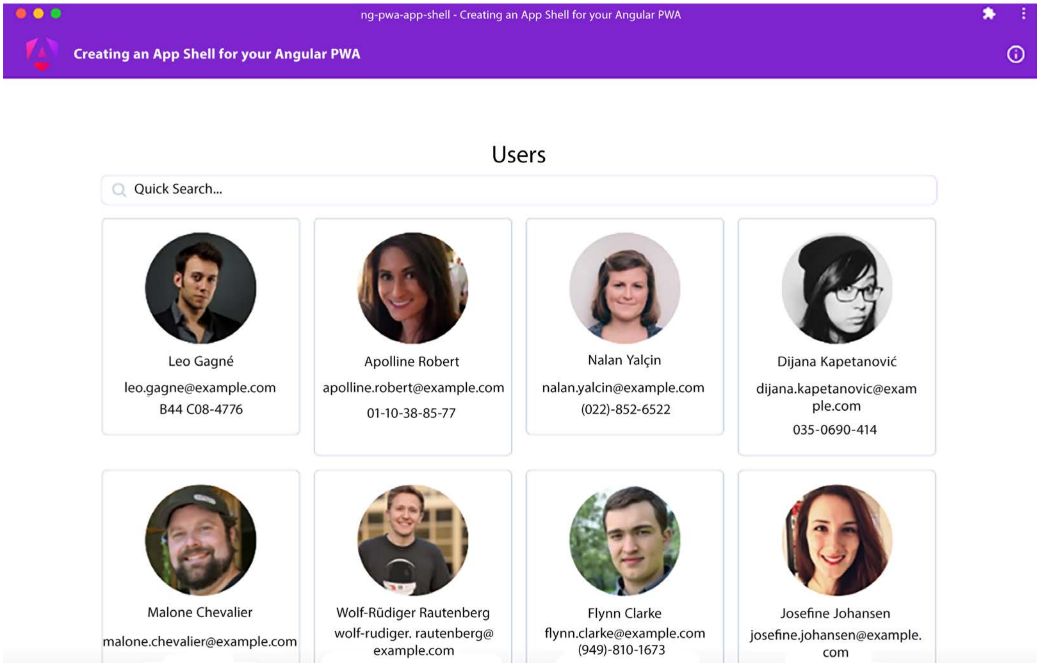
Figure 13.32: App Shell showing the pre-rendered Users page

9. To verify that we are pre-rendering the **Users** page at build time, inspect the generated code at `<workspace-root>/start/dist/apps/chapter13/ng-pwa-app-shell/browser/index.html`. You should see the entire rendered page inside the `<body>` tag.

10. Create the production build with the App Shell and serve it on port `1020` by running the following commands:

```
cd start && nx run ng-pwa-app-shell:app-shell:production
cd ..
npx http-server ./start/dist/apps/chapter13/ng-pwa-app-shell/browser -o
-p 1020
```

11. Navigate to `http://localhost:1020` in your browser and install the app as a PWA, as shown in *Figure 13.8*. Once done, run the PWA and it should look as follows:



ng-pwa-app-shell - Creating an App Shell for your Angular PWA

Creating an App Shell for your Angular PWA

Users

Quick Search...

Leo Gagné
leo.gagne@example.com
B44 C08-4776

Apolline Robert
apolline.robert@example.com
01-10-38-85-77

Nalan Yalçin
nalan.yalcin@example.com
(022)-852-6522

Dijana Kapetanović
dijana.kapetanovic@example.com
035-0690-414

Malone Chevalier
malone.chevalier@example.com

Wolf-Rüdiger Rautenberg
wolf-rudiger. rautenberg@example.com

Flynn Clarke
flynn.clarke@example.com
(949)-810-1673

Josefine Johansen
josefine.johansen@example.com

Figure 13.33: ng-pwa-app-shell running as a PWA

Great! You now know how to create an App Shell for your Angular PWAs. Now that you've finished the recipe, see the next section on how it works.

How it works...

The recipe begins with disabling JavaScript for our application. This means when the app runs, we only show static HTML and CSS since there's no JavaScript execution. We see a message about JavaScript not being supported, as shown in *Figure 13.30*. The code comes from the `src/index.html` file having the following code inside the `<body>` element:

```
<noscript>Please enable JavaScript to continue using this application.</
noscript>
```

We then run the command `nx g app-shell--project=ng-pwa-app-shell`. Since we are in an NX workspace, the command requires the right project. If this were a regular Angular project, you could just run `ng generate app-shell` and it would create the App Shell for you. In either case, the command does the following things for us:

- Creates a new component named `AppShellComponent` and generates its relevant files.
- Installs the `@angular/platform-server` package in the project.
- Adds some new files, namely, `main.server.ts`, to enable server-side rendering (build-time rendering for our App Shell, to be exact).

- Most importantly, it updates the `project.json` (`angular.json` for an Angular project not in the NX workspace) file to add a bunch of schematics for server-side rendering as well as for generating the `app-shell`. Notice that it overrides the `build` object (specifically the `outputPath`) property to remove the `apps` folder and chapter name. We don't want that as we want the bundle to be generated in `dist/apps/chapter13/<project-name>`, and that is why we updated it manually in the recipe. Notice that it also adds a new `"server"` object for SSR and an `"app-shell"` object with configuration to generate the App Shell.

In the recipe, we create the App Shell, and then we import the `UsersComponent` class to the `AppShellComponent` class's `imports` array. Since `AppShellComponent` is a `standalone` component (not part of any `NgModule`), and so is `UsersComponent`, we need to link them together via the `imports` array in `AppShellComponent`. After importing the `UsersComponent` class, we update the `app-shell.component.html` (the App Shell template) to use the `<app-users>` selector, which reflects the `UsersComponent` class. That is the entire **Users** page. Finally, we make sure that we only render the skeleton elements when the Browser is being created (and the same for SSR). Therefore, we use the `PLATFORM_ID` token in the `users.component.ts` file. The token's value contains the name of the platform our application is running on. When the app shell is being generated, it is the server-side environment, and the value of `PLATFORM_ID` is `"server"`. Instead of comparing it with the string `"server"`, i.e., using the condition `this.platformId !== "server"`), we use the `isPlatformBrowser` function from the `@angular/common` package to only fetch the data if the platform is `browser` (which is not the case when generating the app shell).

We then verify the App Shell in *step 4* and *step 5*. These commands generate an Angular build in production mode with the App Shell (non-minified code), and then serve the project on port `4200`, respectively. Note that the `ng-pwa-app-shell` folder in the `start/dist/apps/chapter13/ng-pwa-app-shell` folder has two folders inside, that is, the `browser` folder and the `server` folder, and our `index.html` resides in the `browser` folder. The code inside `index.html` is pre-rendered at build time. This means Angular opens the app and renders the `UsersComponent`, including the search input and skeleton loaders, at build time. So, as soon as the app opens, the content is pre-rendered. Once we're done with all these steps, we install the PWA to test it out.

Now that you know how the recipe works, see the next section for links to further reading.

See also

- Angular App Shell guide: `https://angular.io/guide/app-shell`
- The App Shell model (web fundamentals by Google): `https://developers.google.com/web/fundamentals/architecture/app-shell`
- **Critical Rendering Path** (**CRP**): `https://developer.mozilla.org/en-US/docs/Web/Performance/Critical_rendering_path`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://packt.link/AngularCookbook2e`

`packt.com`

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

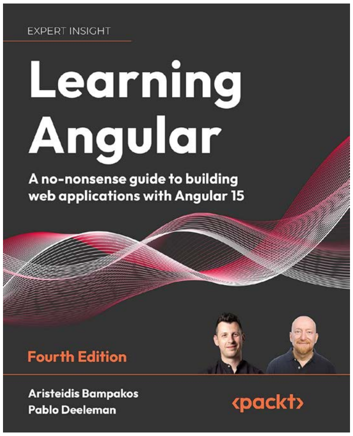
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At `www.packt.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



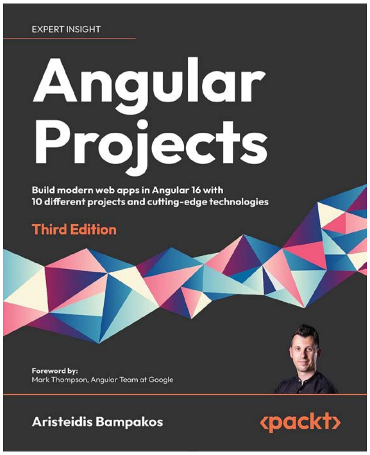
Learning Angular, Fourth Edition

Aristeidis Bampakos

Pablo Deeleman

ISBN: 9781803240602

- Use the Angular CLI to scaffold, build, and deploy a new Angular application
- Build components, the basic building blocks of an Angular application
- Discover new Angular Material components such as Google Maps, YouTube, and multi-select dropdowns
- Understand the different types of templates supported by Angular
- Create HTTP data services to access APIs and provide data to components
- Learn how to build Angular apps without modules in Angular 15.x with standalone APIs
- Improve your debugging and error handling skills during runtime and development

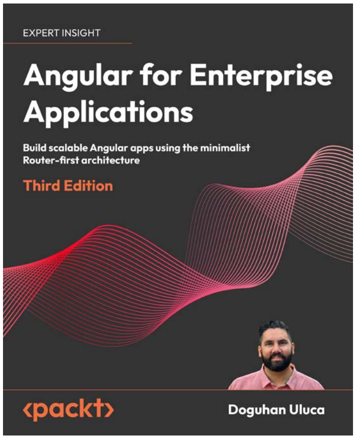


Angular Projects, Third Edition

Aristeidis Bampakos

ISBN: 9781803239118

- Set up Angular applications using Angular CLI and Nx Console
- Create a personal blog with Jamstack, Scully plugins, and SPA techniques
- Build an issue management system using typed reactive forms
- Use PWA techniques to enhance user experience
- Make SEO-friendly web pages with server-side rendering
- Create a monorepo application using Nx tools and NgRx for state management
- Focus on mobile application development using Ionic
- Develop custom schematics by extending Angular CLI


EXPERT INSIGHT
Angular for Enterprise Applications
Build scalable Angular apps using the minimalist Router-first architecture
Third Edition
<packt>
Doguhan Uluca

Angular for Enterprise Applications, Third Edition

Doguhan Uluca

ISBN: 9781805127123

- Adopt a minimalist, value-first approach to delivering web apps
- Understand how standalone components, services, providers, modules, lazy loading, and directives work in Angular
- Learn how to build standalone projects
- Manage your app's data reactivity using Signals or RxJS
- Implement state management for your Angular apps with NgRx
- Learn about root causes of performance bottlenecks of Angular apps
- Learn about the Angular ecosystem to build and deliver enterprise applications
- Leverage automated testing and CI/CD to deliver high quality apps
- Master authentication and authorization building role-based access control with REST and GraphQL

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Angular Cookbook - Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

@Input() bindings 405

***ngFor**
  trackBy, using for lists with 425-429

@ngrx/component-store
  reference link 195

@ngrx/component-store, effect
  reference link 195

A

accessible listbox interactions
  with CDK Listbox directives 307-310

actions
  NgRx store, creating with 171-176

aliased class providers
  services, providing with 95- 97

Angular
  lazily loaded routes 201-204

Angular animations
  disabling conditionally 132-134
  using 111

Angular App performance, boosting with performance budgets
  reference link 447

Angular App Shell guide
  reference link 490

Angular CanActivateFn docs
  reference link 214

Angular CDK
  input coercion utilities from 284-287

Angular CDK accessibility documentation
  reference link 272

Angular CDK component harnesses
  used, for writing tests 344-349

Angular CDK Drag and Drop
  reference link 295

Angular CDK Listbox APIs 307

Angular CDK Menu
  reference link 318

Angular CDK Menu API
  used, for working with nested menus 310-318

Angular CDK Overlay API
  reference link 283

Angular CDK scrolling examples
  reference link 267

Angular CDK Stepper examples
  reference link 307

Angular CLI, moving to jest
  reference link 325

Angular command-line interface (Angular CLI) 263

Angular Component Dev Kit (Angular CDK) 263

Angular DI tokens
  using 79-81

Angular FormArray
  reference link 252

Angular pipes
  unit-testing, writing for 355-358

Angular pipes, testing
  reference link 358

Angular pure and impure pipes
  reference link 436

Angular Reactive Extensions 171

Angular router docs
  reference link 201

Angular services
  mocking, with stubs 330-334
Angular service worker
  precache network requests with 480-483
Angular service worker config
  reference link 483
Angular service worker introduction
  reference link 460, 483
Angular (standalone) app
  routes, creating 197-201
Angular standalone components
  reference link 201
Angular testing component scenarios
  reference link 355
Angular v16
  migrating, to Jest 324
API calls
  NgRx effects, used for fetching data from 183-189
API for component harness authors
  reference link 350
appReadTime directive 56
  creating 50-55
appScrollTo directive 58
App Shell
  creating, for Progressive Web Apps (PWAs) 484-490
App Shell model, architecture
  reference link 490
async events
  running, with runOutsideAngular 416-425
asynchronous validator functions
  used, for server-side validation 242-246
asynchronous validators 242
attribute directives
  creating, for read time calculation for articles 50-56
  creating, for vertical element scrolling 56-59
  used, for handling element appearance 44-49
auditing
  performance budgets, using for 443-446
authorized access, to routes
  using route guards 212-214

B

BehaviorSubject 10
Bind Route Info to Component Inputs
  reference link 219
bundles
  analyzing, with webpack-bundle-analyzer 447-450

C

CDK coercion documentation
  reference link 287
CDK Drag and Drop API
  using, to move items from one list to another 287-295
CDK Listbox
  reference link 310
CDK Listbox directives
  used, for accessible listbox interactions 307-310
CdkOverlayOrigin directive
  reference link 283
CDK Stepper API
  used, for creating multi-step game 295-306
ChangeDetectionStrategy
  reference link 405
change detector
  detaching, from components 406-415
ChangeDetectorRef
  reference link 415, 425

child component
  accessing in parent component class, ViewChild used 24-26
  accessing in parent template, via template variables 21-23
child components and directives
  mocking, with ng-mocks package 337-343
Command-line Interface (CLI) 441
complex Angular animations
  creating, keyframes used 111-116
complex forms
  implementing, with Reactive Form Arrays 246-252
component communication
  establishing, @Input and @Output properties, used 2-6
  establishing, services used 7-11
  establishing, signals used 31-41
component @Input and @Output properties
  using, for component communication 2-6
components
  change detector, detaching from 406-415
component subtrees
  pruning, with OnPush change detection 400-405
component testing scenarios
  reference link 334
Content Delivery Network (CDN) 389
ControlValueAccessor
  custom form control, writing with 252-262
Critical Rendering Path (CRP)
  reference link 490
CSS selector 349
custom form control
  writing, with ControlValueAccessor 252-262
custom installable experience
  providing, in PWA 472-480
custom RxJS operator
  creating 163-166
custom structural directive
  writing 60-66
Cypress.$ 389
Cypress bundled libraries
  reference link 389
Cypress bundled packages
  using 382-389
Cypress fixtures
  using, to provide mock data 389-397
cypress-real-events project repository
  reference link 368
Cypress test
  writing 359-363

D

dark mode
  using, in PWA 463-471
data 295
data transformation
  map operator, using 155-159
debounceTime operator
  using, with autocompletes 159-163
DI in Angular
  reference link 81
Directive Composition API 43, 76
  used, for applying multiple directives to same element 70-75
Document Object Model (DOM) 43, 264
DOM element
  validating, for visibility on view 363-368
dumb components 405
dynamic configurations
  value providers, using for 97-100

E

element appearance
  handling, with attribute directives 44-49
Element.scrollIntoView() 60
elements, finding in component's DOM
  reference link 349
End-to-End (E2E) tests 359
  reference link 363
existing Angular app
  converting, into PWAs
    with Angular CLI 453-460
exponential backoff strategy 167
exponential backoff technique 166

F

failed HTTP calls
  retrying, with RxJS 166-169
fixtures
  reference link 397
form control 146
form inputs and submission
  testing 368-373
forRoot()
  used, for creating singleton service 91-95

G

get
  reference link 363
global loader
  displaying, between route transition 219-222
global mocks
  providing, for Jest 325-330

H

heavy computation
  moving, to pure Angular pipes 429-436
  web workers, using for 436-443
hierarchical injectors
  reference link 86, 97
host element 66
HTTP calls 405
  unit-tests, creating with 350-355
HttpTestingController docs
  reference link 355
huge lists
  virtual scroll, using for 264-267

I

in-app install experience
  reference link 480
InjectionToken documentation
  reference link 81
input coercion utilities
  from Angular CDK 284-287
input property changes
  intercepting, ngOnChanges used 16-19
  intercepting, setters used 11-16
intercept
  reference link 397
items
  moving, from one list to another with CDK Drag and Drop API 287-295

J

Jest 319
  global mocks, providing for 325-330
  reference link 325
  used, for setting up unit tests in Angular 320-324

Jest documentation for setupFiles
  reference link 330
jest.fn
  reference link 334
jQuery
  URL 389
JSDOM (JavaScript Document Object Model) 329

K

keyboard navigation
  for lists 267-272
keyframes
  used, for creating complex Angular animations 111-116

L

lazily loaded routes
  in Angular 201-204
  reference link 205
lazily loading standalone components
  reference link 205
lifecycle hooks 20
Listbox Pattern
  reference link 310
lists
  animating, with stagger animations 116-119
  keyboard navigation 267-272
lodash
  URL 389

M

Manual Mocks with Jest
  reference link 330
map operator
  used, for transforming data 155-159
markForCheck method
  reference link 405
MDN web worker
  reference link 443
mock data
  providing, with Cypress fixtures 389-397
ModuleWithProviders Angular documentation
  reference link 95
ModuleWithProviders migration documentation
  reference link 95
multiple observable streams
  listening 142-146
  working 146
multiple structural directives
  applying, to same element 67-70
  applying, to same element with Directive Composition API 70-75
multi-state animations
  working with 107-111
multi-step game
  creating, with CDK Stepper API 295-306

N

nested menus
  working, with Angular CDK Menu API 310-318
NgForOf
  reference link 429
ng-mocks documentation
  reference link 343
ng-mocks package
  used, for mocking child components and directives 337-343
ng-mocks package's mock method
  reference link 343
ngOnChanges
  used, for intercepting input property changes 16-19
  working 20

NgRx 171

NgRx actions
  reference link 176

NgRx Component Store
  used, for managing state of component 190-195

NgRx effects
  reference link 189
  used, for fetching data from API calls 183-189

NgRx reducers
  reference link 176

NgRx selectors
  reference link 183
  used, for rendering state in components 180-183
  used, for selecting state in components 180-183

NgRx store
  creating, with actions 171-176
  creating, with reducers 171-176
  reference link 176

NgRx Store Devtools
  reference link 179
  using, to debug state changes 176-179

NgZone
  reference link 425

O

Observable 10

offline fallback page creation
  reference link 483

OnPush change detection
  using, to prune component subtrees 400-405

optional dependencies 81, 82
  reference link 86
  using 82-86
  working 86

Overlay API
  pointy little popovers with 273-283

P

Page Objects (POs) 348, 363

Pages 348

parallel animations
  versus sequential animations 121-125

parent component class
  child component, accessing with ViewChild 24-26

parent template
  child component, accessing via template variables 21-23

performance budgets
  using, for auditing 443-446

performance budgets with Angular CLI
  reference link 447

precache network requests
  with Angular service worker 480-483

prefers-color-scheme
  reference link 471

prefers-color-scheme user preference media feature
  reference link 471

presentational components 405

Progressive Web Apps (PWAs) 453
  App Shell, creating for 484-490
  custom installable experience, providing 472-480
  dark mode, using 463-471
  reference link 460
  theme color, modifying 460-463

providedIn
  used, for creating singleton service 86-91

pure Angular pipes
  heavy computation, moving to 429-436

PWAs with Angular CLI
  existing Angular app, converting into 453-460
  reference link 463

Q

query parameters and fragments, accessing
  reference link 219

R

Reactive form
  creating, with form validation 229-235
Reactive Form Arrays
  complex forms, implementing with 246-252
Reactive Forms API 143
reactive programming 171
reducers
  NgRx store, creating with 171-176
Request for Comments (RFC) 40
retry operator 170
route animations
  implementations 125-132
route guards
  used, for authorized access to routes 212-214
route information
  reference link 219
route parameters
  working with 215-219
route preloading strategies 205-211
  references 212
router events docs
  reference link 222
router service docs
  reference link 222
routes
  creating, in Angular (standalone) app 197-201
route transition
  global loader, displaying between 219-222
runOutsideAngular
  used, for running async events outside Angular 416-425

S

scroll-to directive
  creating 57
sequential and parallel HTTP calls
  making, with RxJS 137-141
sequential animations
  versus parallel animations 121-125
server-side validation
  asynchronous validator functions, using for 242-246
services
  using, for component communication 7-10
setters
  used, for intercepting input property changes 11-16
signals 31
  using, for component communication 31-41
Single Page Application (SPA) 222
singleton service
  creating, using providedIn 86-91
  creating, with forRoot() 91-95
spies
  using, on injected service in unit test 334-337
stagger animations 120
  list, animating with 116-120
standalone components
  data, passing through route params 26-30
  working with 26
Star Wars API (Swapi) 137
state changes
  debugging, with NgRx Store Devtools 176-179
state of component
  managing, with NgRx Component Store 190-195
  rendering and selecting, with NgRx selector 180-183

streams 146
  unsubscribing automatically, async pipe used 152-155
  unsubscribing, to avoid memory leaks 146-148
  working 149-151
structural directives 66, 67
stubs
  used, for mocking Angular services 330-334
switchMap operator
  using, with autocompletes 159-163

T

template-driven form
  creating, with validation 223-229
  reference link 229
  testing, with unit tests 235-242
template-driven form, testing
  reference link 242
template reference variable 23
template variables
  for accessing child component, in parent template 21-23
Test Angular Pipes With Services
  reference link 358
Testing Angular Services
  reference link 337
tests
  writing, with Angular CDK component harnesses 344-349
theme color
  modifying, for PWA 460-463
trackBy
  using, for lists with *ngFor 425-429
two-state Angular animation
  creating 103-107

U

unauthorized access, preventing in Angular routes
  reference link 214
unit tests
  creating, with HTTP calls 350-355
  setting up, in Angular with Jest 320-324
  spies, using on injected service 334-337
  template-driven form, testing with 235-242
  writing, for Angular Pipes 355-358
User Experience (UX) 159, 294, 400
User Interface (UI) 374

V

value providers
  used, for dynamic configurations 97-100
variable aliasing 381
ViewChild
  reference link 272
  used, for accessing child component in parent component class 24-26
view query 26
virtual scroll
  using, for huge lists 264-267
visibility of items
  reference link 368

W

WAI ARIA pattern for listboxes
  reference link 310
wait command
  reference link 382
Web Frameworks, testing with Jest
  reference link 325
webpack
  reference link 450

webpack-bundle-analyzer
  used, for analyzing bundles 447-450
web workers
  reference link 443
  using, for heavy computation 436-443

X

XMLHttpRequests (XHRs)
  waiting, to finish 374-382

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don’t worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don’t stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/9781803233444

2. Submit your proof of purchase
3. That’s it! We’ll send your free PDF and other benefits to your email directly